To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

User's Manual

MR8C/4 V.4.00

User's Manual

Real-time OS for R8C Family

Renesas Electronics
www.renesas.com

Rev.1.00 2009.08

- Active X, Microsoft, MS-DOS, Visual Basic, Visual C++, Windows and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and other countries.
- IBM and AT are registered trademarks of International Business Machines Corporation.
- Intel and Pentium are registered trademarks of Intel Corporation.
- Adobe, Acrobat, and Acrobat Reader are trademarks of Adobe Systems Incorporated.
- TRON is an abbreviation of "The Real-time Operating system Nucleus."
- ITRON is an abbreviation of "Industrial TRON."
- μITRON is an abbreviation of "Micro Industrial TRON."
- TRON, ITRON, and μITRON do not refer to any specific product or products.
- All other brand and product names are trademarks, registered trademarks or service marks of their respective holders.

Keep safety first in your circuit designs!

- Renesas Technology Corporation and Renesas Solutions Corporation put the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage. Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

- These materials are intended as a reference to assist our customers in the selection of the Renesas Technology product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation, Renesas Solutions Corporation or a third party.
- Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
- All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation and Renesas Solutions Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation, Renesas Solutions Corporation or an authorized Renesas Technology product distributor for the latest product information before purchasing a product listed herein. The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors. Please also pay attention to information published by Renesas Technology Corporation and Renesas Solutions Corporation by various means, including the Renesas home page (http://www.renesas.com).
- When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, liability or other loss resulting from the information contained herein.
- Renesas Technology semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation, Renesas Solutions Corporation or an authorized Renesas Technology product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
- The prior written approval of Renesas Technology Corporation and Renesas Solutions Corporation is necessary to reprint or reproduce in whole or in part these materials.
- If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination. Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
- Please contact Renesas Technology Corporation or Renesas Solutions Corporation for further details on these materials or the products contained therein.

For inquiries about the contents of this document or product, fill in the text file the installer generates in the following directory and email to your local distributor.

\SUPPORT\Product-name\SUPPORT.TXT

Renesas Tools Homepage http://www.renesas.com/en/tools

Preface

The MR8C/4 is a real-time operating system[1] for R8C Family microcomputers. The MR8C/4 conforms to the μITRON Specification.[2]

This manual describes the procedures and precautions to observe when you use the MR8C/4 for programming purposes. For the detailed information on individual service call procedures, refer to the MR8C/4 Reference Manual.

Requirements for MR8C/4 Use

When creating programs based on the MR8C/4, it is necessary to purchase the following product of Renesas.

- C-compiler package M3T-NC30WA(abbreviated as NC30) for the M16CSeries and R8C Family micro-computers.

Document List

The following sets of documents are supplied with the MR8C/4.

- Release Note
  Presents a software overview and describes the corrections to the Users Manual and Reference Manual.
- Users Manual (PDF file)
  Describes the procedures and precautions to observe when using the MR8C/4 for programming purposes.

Right of Software Use

The right of software use conforms to the software license agreement. You can use the MR8C/4 for your product development purposes only, and are not allowed to use it for the other purposes. You should also note that this manual does not guarantee or permit the exercise of the right of software use.

[1] Hereinafter abbreviated "real-time OS"

[2] μITRON4.0 Specification is the open real-time kernel specification upon which the TRON association decided
The specification document of μITRON4.0 specification can come to hand from a TRON association homepage (http://www.assoc.tron.org/).

Contents

List of Figures

List of Tables

1. User's Manual Organization

The MR8C/4 User's Manual consists of nine chapters and thee appendix.

- 2 General Information
  Outlines the objective of MR8C/4 development and the function and position of the MR8C/4.
- 3 Introduction to Kernel
  Explains about the ideas involved in MR8C/4 operations and defines some relevant terms.
- 4 Kernel
  Outlines the applications program development procedure for the MR8C/4.
- 5 Service call reference
  Details MR8C/4 service call API.
- 6 Applications Development Procedure Overview
  Details the applications program development procedure for the MR8C/4.
- 7 Detailed Applications
  Presents useful information and precautions concerning applications program development with MR8C/4.
- 8 Using Configurator
  Describes the method for writing a configuration file and the method for using the configurator in detail.
- 9 Sample Program Description
  Describes the MR8C/4 sample applications program which is included in the product in the form of a source file.
- 10 Stack Size Calculation Method
  Describes the calculation method of the task stack size and the system stack size.
- 11 Note
  Presents useful information and precautions concerning applications program development with MR8C/4.
- 12 Appendix
  Data type and assembly language interface.

2. General Information

2.1 Objective of MR8C/4 Development

In line with recent rapid technological advances in microcomputers, the functions of microcomputer-based products have become complicated. In addition, the microcomputer program size has increased. Further, as product development competition has been intensified, manufacturers are compelled to develop their microcomputer-based products within a short period of time.

In other words, engineers engaged in microcomputer software development are now required to develop larger-size programs within a shorter period of time. To meet such stringent requirements, it is necessary to take the following considerations into account.

1. **To enhance software recyclability to decrease the volume of software to be developed.**

   One way to provide for software recyclability is to divide software into a number of functional modules wherever possible. This may be accomplished by accumulating a number of general-purpose subroutines and other program segments and using them for program development. In this method, however, it is difficult to reuse programs that are dependent on time or timing. In reality, the greater part of application programs are dependent on time or timing. Therefore, the above recycling method is applicable to only a limited number of programs.

2. **To promote team programming so that a number of engineers are engaged in the development of one software package**

   There are various problems with team programming. One major problem is that debugging can be initiated only when all the software program segments created individually by team members are ready for debugging. It is essential that communication be properly maintained among the team members.

3. **To enhance software production efficiency so as to increase the volume of possible software development per engineer.**

   One way to achieve this target would be to educate engineers to raise their level of skill. Another way would be to make use of a structured descriptive assembler, C-compiler, or the like with a view toward facilitating programming. It is also possible to enhance debugging efficiency by promoting modular software development.

However, the conventional methods are not adequate for the purpose of solving the problems. Under these circumstances, it is necessary to introduce a new system named real-time OS [3]

To answer the above-mentioned demand, Renesas has developed a real-time operating system, tradenamed MR8C/4, for use with the R8C Family of 16-bit microcomputers .

When the MR8C/4 is introduced, the following advantages are offered.

1. **Software recycling is facilitated.**

   When the real-time OS is introduced, timing signals are furnished via the real-time OS so that programs dependent on timing can be reused. Further, as programs are divided into modules called tasks, structured programming will be spontaneously provided.
   That is, recyclable programs are automatically prepared.

2. **Ease of team programming is provided.**

   When the real-time OS is put to use, programs are divided into functional modules called tasks. Therefore, engineers can be allocated to individual tasks so that all steps from development to debugging can be conducted independently for each task.
   Further, the introduction of the real-time OS makes it easy to start debugging some already finished tasks even if the entire program is not completed yet. Since engineers can be allocated to individual tasks, work assignment is easy.

3. **Software independence is enhanced to provide ease of program debugging.**

   As the use of the real-time OS makes it possible to divide programs into small independent modules called tasks,

[3] OS:Operating System

the greater part of program debugging can be initiated simply by observing the small modules.

4. Timer control is made easier.

To perform processing at 10 ms intervals, the microcomputer timer function was formerly used to periodically initiate an interrupt. However, as the number of usable microcomputer timers was limited, timer insufficiency was compensated for by, for instance, using one timer for a number of different processing operations.
When the real-time OS is introduced, however, it is possible to create programs for performing processing at fixed time intervals making use of the real-time OS time management function without paying special attention to the microcomputer timer function. At the same time, programming can also be done in such a manner as to let the programmer take that numerous timers are provided for the microcomputer.

5. Software maintainability is enhanced

When the real-time OS is put to use, the developed software consists of small program modules called tasks. Therefore, increased software maintainability is provided because developed software maintenance can be carried out simply by maintaining small tasks.

6. Increased software reliability is assured.

The introduction of the real-time OS makes it possible to carry out program evaluation and testing in the unit of a small module called task. This feature facilitates evaluation and testing and increases software reliability.

7. The microcomputer performance can be optimized to improve the performance of microcomputer-based products.

With the real-time OS, it is possible to decrease the number of unnecessary microcomputer operations such as I/O waiting. It means that the optimum capabilities can be obtained from microcomputers, and this will lead to microcomputer-based product performance improvement.

2.2 Relationship between TRON Specification and MR8C/4

MR8C/4 is the real-time operating system developed for use with the R8C Family of 16-bit microcomputers compliant with µITRON 4.0 Specification. µITRON 4.0 Specification stipulates standard profiles as an attempt to ensure software portability. Of these standard profiles, MR8C/4 has implemented in it many useful service calls.

2.3 Features

The MR8C/4 offers the following features.

1. **Real-time operating system conforming to the μITORN Specification.**

   The MR8C/4 is designed in compliance with the μITRON Specification which incorporates a minimum of the ITRON Specification functions so that such functions can be incorporated into a one-chip microcomputer. As the μITRON Specification is a subset of the ITRON Specification, most of the knowledge obtained from published ITRON textbooks and ITRON seminars can be used as is.
   Further, the application programs developed using the real-time operating systems conforming to the ITRON Specification can be transferred to the MR8C/4 with comparative ease.

2. **High-speed processing is achieved.**

   MR8C/4 enables high-speed processing by taking full advantage of the microcomputer architecture.

3. **Only necessary modules are automatically selected to constantly build up a system of the minimum size.**

   MR8C/4 is supplied in the object library format of the R8C Family.
   Therefore, the Linkage Editor LN30 functions are activated so that only necessary modules are automatically selected from numerous MR8C/4 functional modules to generate a system.
   Thanks to this feature, a system of the minimum size is automatically generated at all times.

4. **An upstream process tool named "Configurator" is provided to simplify development procedures**

   A configurator is furnished so that various items including a ROM write form file can be created by giving simple definitions.
   Therefore, there is no particular need to care what libraries must be linked.
   In addition, a GUI version of the configurator is available. It helps the user to create a configuration file without the need to learn how to write it.

3. Introduction to Kernel

3.1 Concept of Real-time OS

This section explains the basic concept of real-time OS.

3.1.1 Why Real-time OS is Necessary

In line with the recent advances in semiconductor technologies, the single-chip microcomputer ROM capacity has increased. ROM capacity of 32K bytes.

As such large ROM capacity microcomputers are introduced, their program development is not easily carried out by conventional methods. Figure 3.1 shows the relationship between the program size and required development time (program development difficulty).

This figure is nothing more than a schematic diagram. However, it indicates that the development period increases exponentially with an increase in program size.

For example, the development of four 8K byte programs is easier than the development of one 32K byte program.[4]

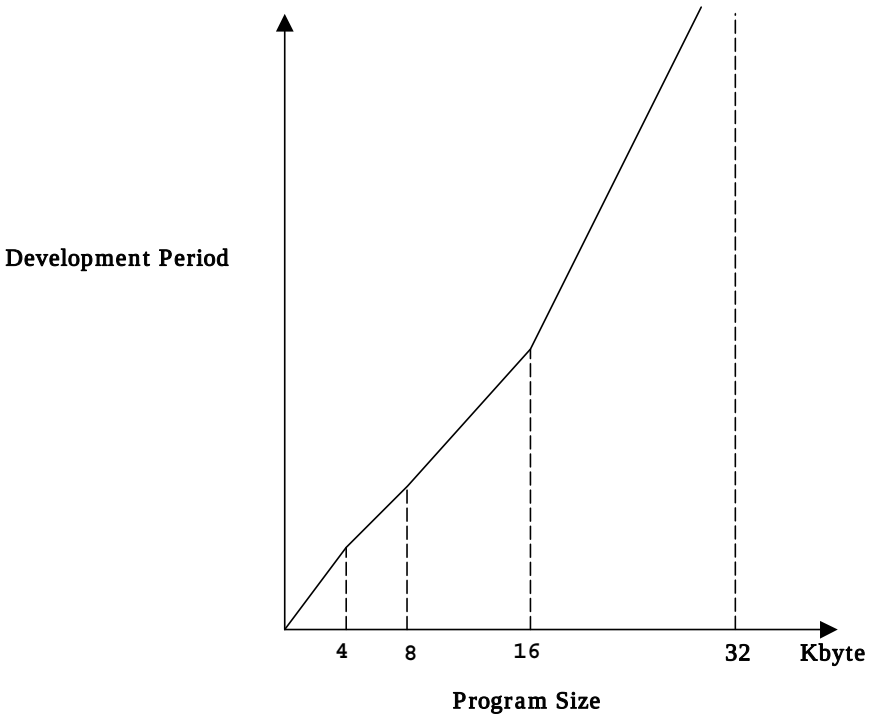


Figure 3.1 Relationship between Program Size and Development Period

Under these circumstances, it is necessary to adopt a method by which large-size programs can be developed within a short period of time. One way to achieve this purpose is to use a large number of microcomputers having a small ROM capacity. Figure 3.2 presents an example in which a number of microcomputers are used to build up an audio equipment system.

[4] On condition that the ROM program burning step need not be performed.

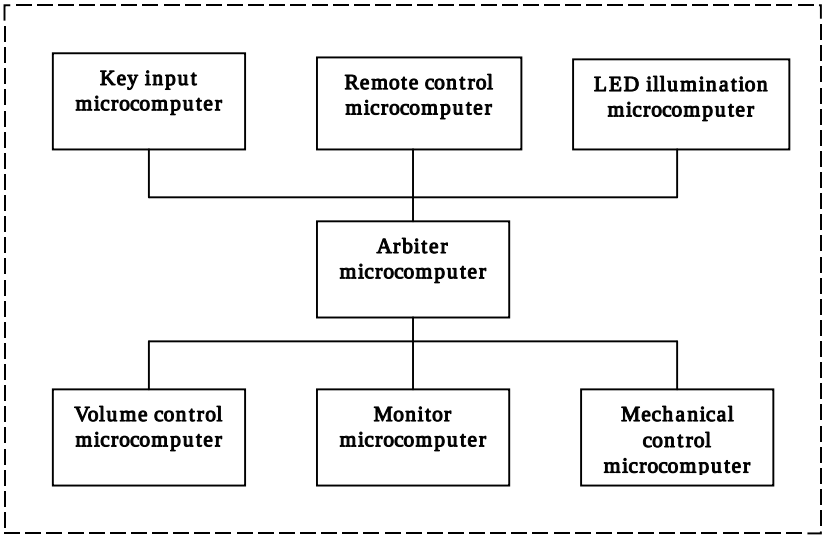


Figure 3.2 Microcomputer-based System Example(Audio Equipment)

Using independent microcomputers for various functions as indicated in the above example offers the following advantages.

1. **Individual programs are small so that program development is easy.**
2. **It is very easy to use previously developed software.**
3. **Completely independent programs are provided for various functions so that program development can easily be conducted by a number of engineers.**

On the other hand, there are the following disadvantages.

1. **The number of parts used increases, thereby raising the product cost.**
2. **Hardware design is complicated.**
3. **Product physical size is enlarged.**

Therefore, if you employ the real-time OS in which a number of programs to be operated by a number of microcomputers are placed under software control of one microcomputer, making it appear that the programs run on separate microcomputers, you can obviate all the above disadvantages while retaining the above-mentioned advantages.

Figure 3.3 shows an example system that will be obtained if the real-time OS is incorporated in the system indicated in Figure 3.2.

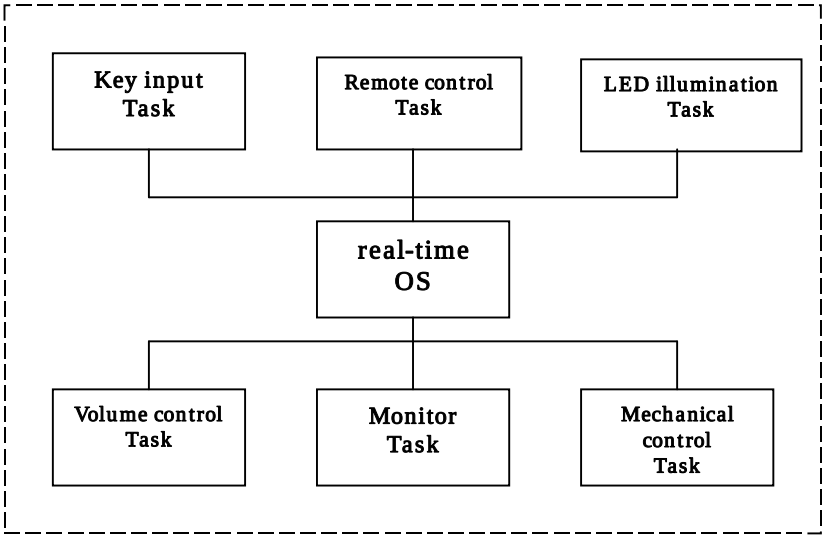


Figure 3.3 Example System Configuration with Real-time OS(Audio Equipment)

In other words, the real-time OS is the software that makes a one-microcomputer system look like operating a number of microcomputers.

In the real-time OS, the individual programs, which correspond to a number of microcomputers used in a conventional system, are called tasks.

3.1.2 Operating Principles of Kernel

A kernel is the core program of real-time OS. The kernel is the software that makes a one-microcomputer system look like operating a number of microcomputers. You should be wondering how the kernel makes a one-microcomputer system function like a number of microcomputers.

As shown in Figure 3.4 the kernel runs a number of tasks according to the time-division system. That is, it changes the task to execute at fixed time intervals so that a number of tasks appear to be executed simultaneously.

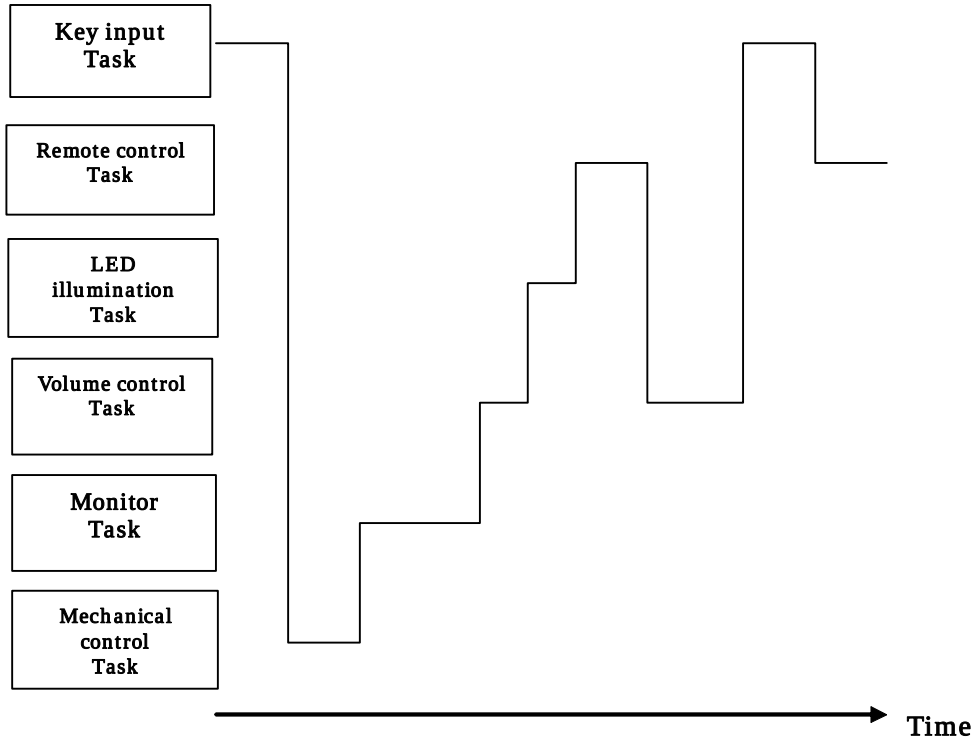


Figure 3.4 Time-division Task Operation

As indicated above, the kernel changes the task to execute at fixed time intervals. This task switching may also be referred to as dispatching. The factors causing task switching (dispatching) are as follows.

- Task switching occurs upon request from a task.
- Task switching occurs due to an external factor such as interrupt.

When a certain task is to be executed again upon task switching, the system resumes its execution at the point of last interruption (See Figure 3.5).

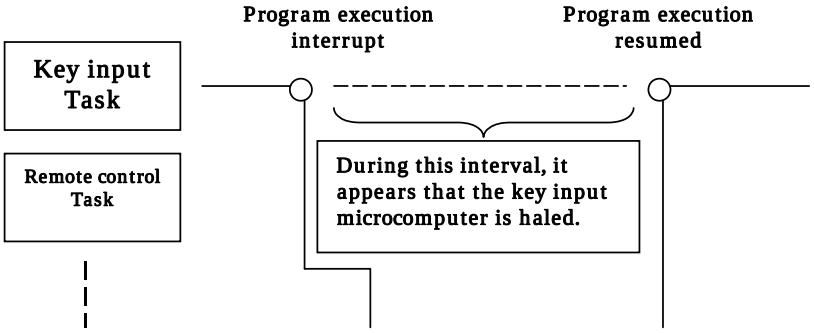


Figure 3.5 Task Execution Interruption and Resumption

In the state shown in Figure 3.5, it appears to the programmer that the key input task or its microcomputer is halted while another task assumes execution control.

Task execution restarts at the point of last interruption as the register contents prevailing at the time of the last interruption are recovered. In other words, task switching refers to the action performed to save the currently executed task register contents into the associated task management memory area and recover the register contents for the task to switch to.

To establish the kernel, therefore, it is only necessary to manage the register for each task and change the register contents upon each task switching so that it looks as if a number of microcomputers exist (See Figure 3.6).

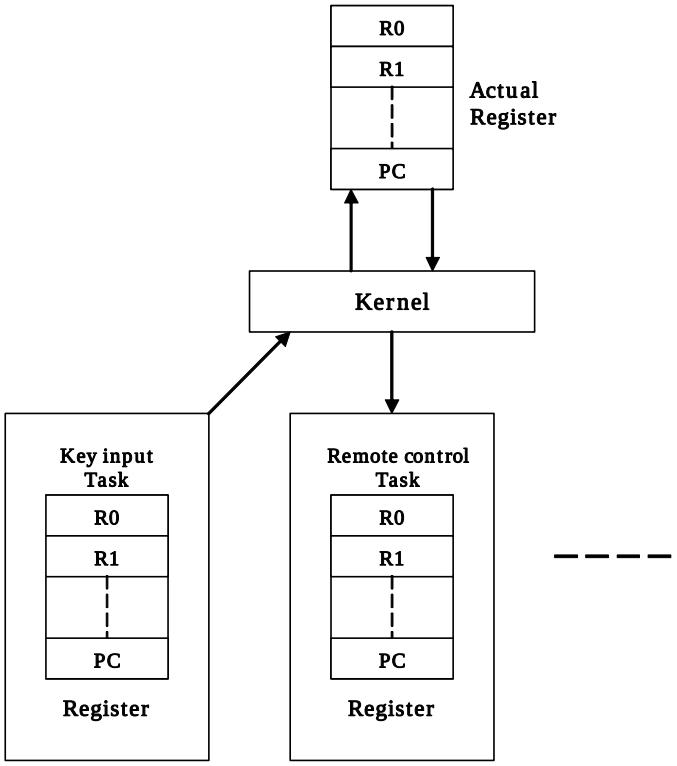


Figure 3.6 Task Switching

The example presented in Figure 3.7 [5] indicates how the individual task registers are managed. In reality, it is necessary to provide not only a register but also a stack area for each task.

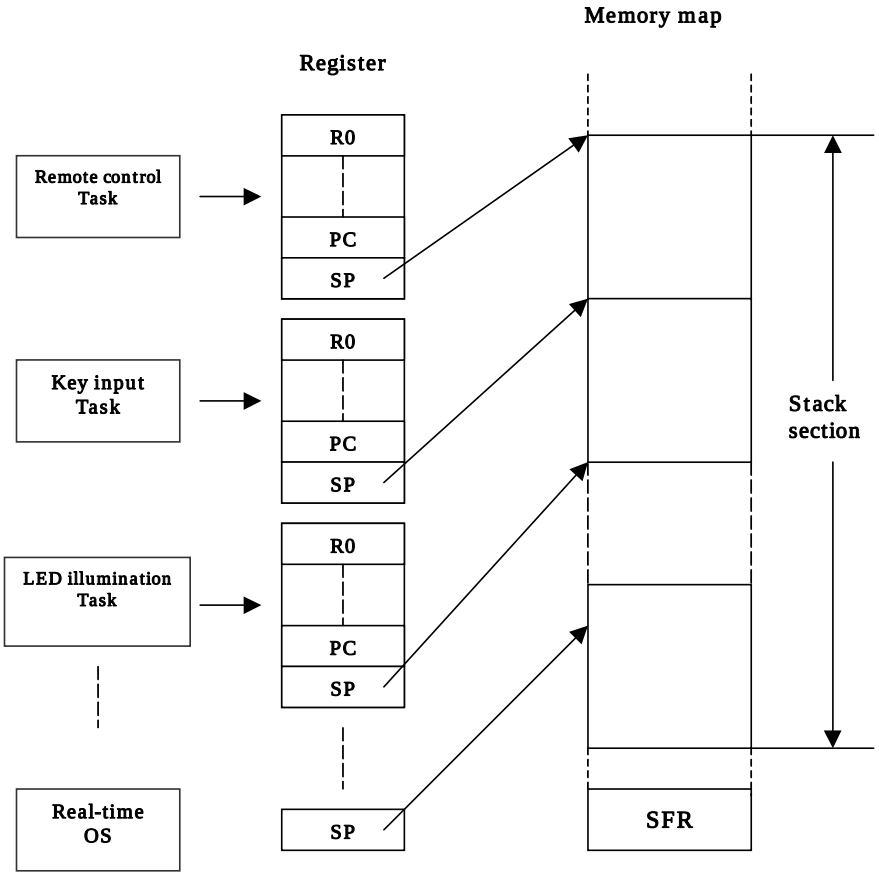


Figure 3.7 Task Register Area

Figure 3.8 shows the register and stack area of one task in detail. In the MR8C/4, the register of each task is stored in a stack area as shown in Figure 3.8. This figure shows the state prevailing after register storage.

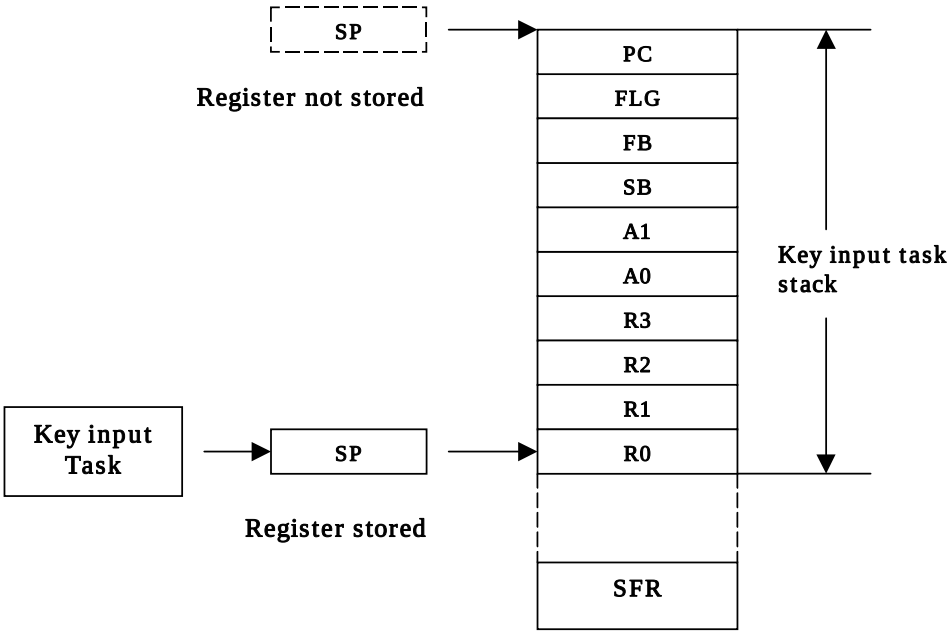


Figure 3.8 Actual Register and Stack Area Management

3.2 Service Call

How does the programmer use the kernel functions in a program?

First, it is necessary to call up kernel function from the program in some way or other. Calling a kernel function is referred to as a service call. Task activation and other processing operations can be initiated by such a service call (See Figure 3.9).

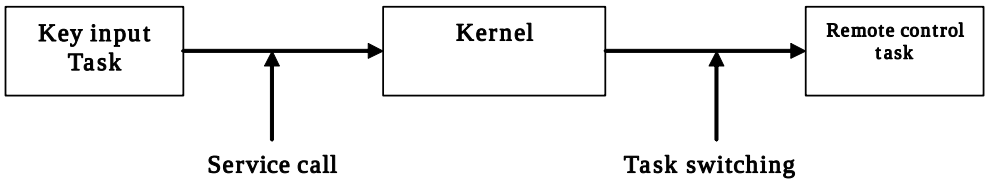


Figure 3.9 Service call

This service call is realized by a function call when the application program is written in C language, as shown below.

```
sta_tsk(ID_main,3);
```

Furthermore, if the application program is written in assembly language, it is realized by an assembler macro call, as shown below.

```
sta_tsk #ID_main,3
```

3.2.1 Service Call Processing

When a service call is issued, processing takes place in the following sequence.[6]

1. **The current register contents are saved.**
2. **The stack pointer is changed from the task type to the real-time OS (system) type.**
3. **Processing is performed in compliance with the request made by the service call.**
4. **The task to be executed next is selected.**
5. **The stack pointer is changed to the task type.**
6. **The register contents are recovered to resume task execution.**

The flowchart in Figure 3.10 shows the process between service call generation and task switching.

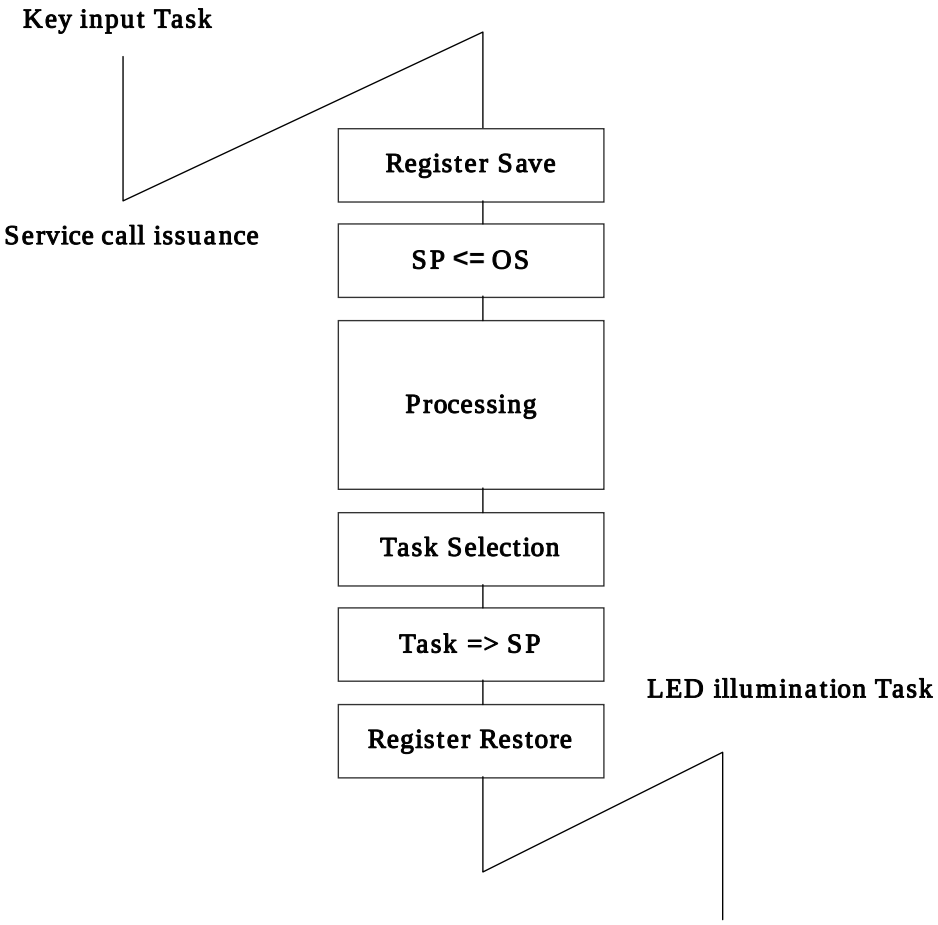


Figure 3.10 Service Call Processing Flowchart

[6] A different sequence is followed if the issued service call does not evoke task switching.

3.2.2 Processing Procedures for Service Calls from Handlers

When a service call is issued from a handler, task switching does not occur unlike in the case of a service call from a task. However, task switching occurs when a return from a handler [7] is made.

The processing procedures for service calls from handlers are roughly classified into the following three types.

1. **A service call from a handler that caused an interrupt during task execution**
2. **A service call from a handler that caused an interrupt during service call processing**
3. **A service call from a handler that caused an interrupt (multiplex interrupt) during handler execution**

[7] The service call can't be issued from OS-independent handler. Therefore, The handler described here does not include the OS-independent handler.

Service Calls from a Handler That Caused an Interrupt during Task Execution
Scheduling (task switching) is initiated by the ret_int service call [8](See Figure 3.11).

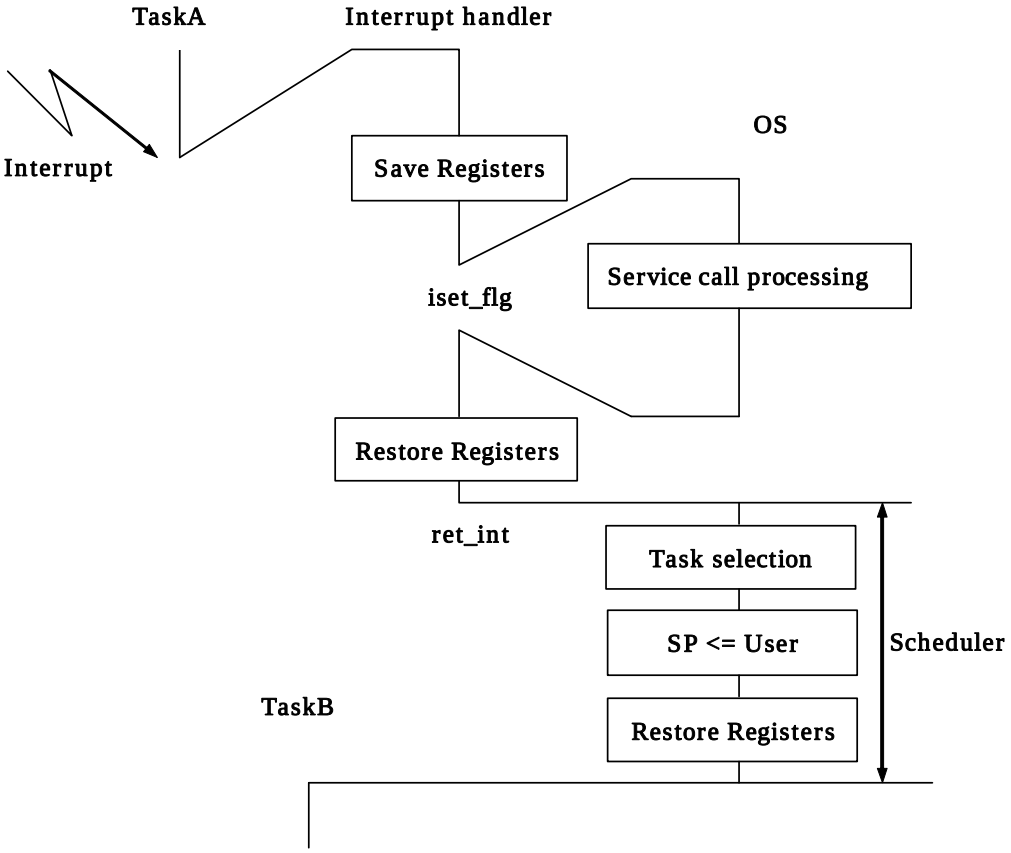


Figure 3.11 Processing Procedure for a Service Call a Handler that caused an interrupt during Task Execution

[8] The ret_int service call is issued automatically when OS-dependent handler is written in C language (when #pragma INTHANDLER specified)

Service Calls from a Handler That Caused an Interrupt during Service Call Processing

Scheduling (task switching) is initiated after the system returns to the interrupted service call processing (See Figure 3.12).

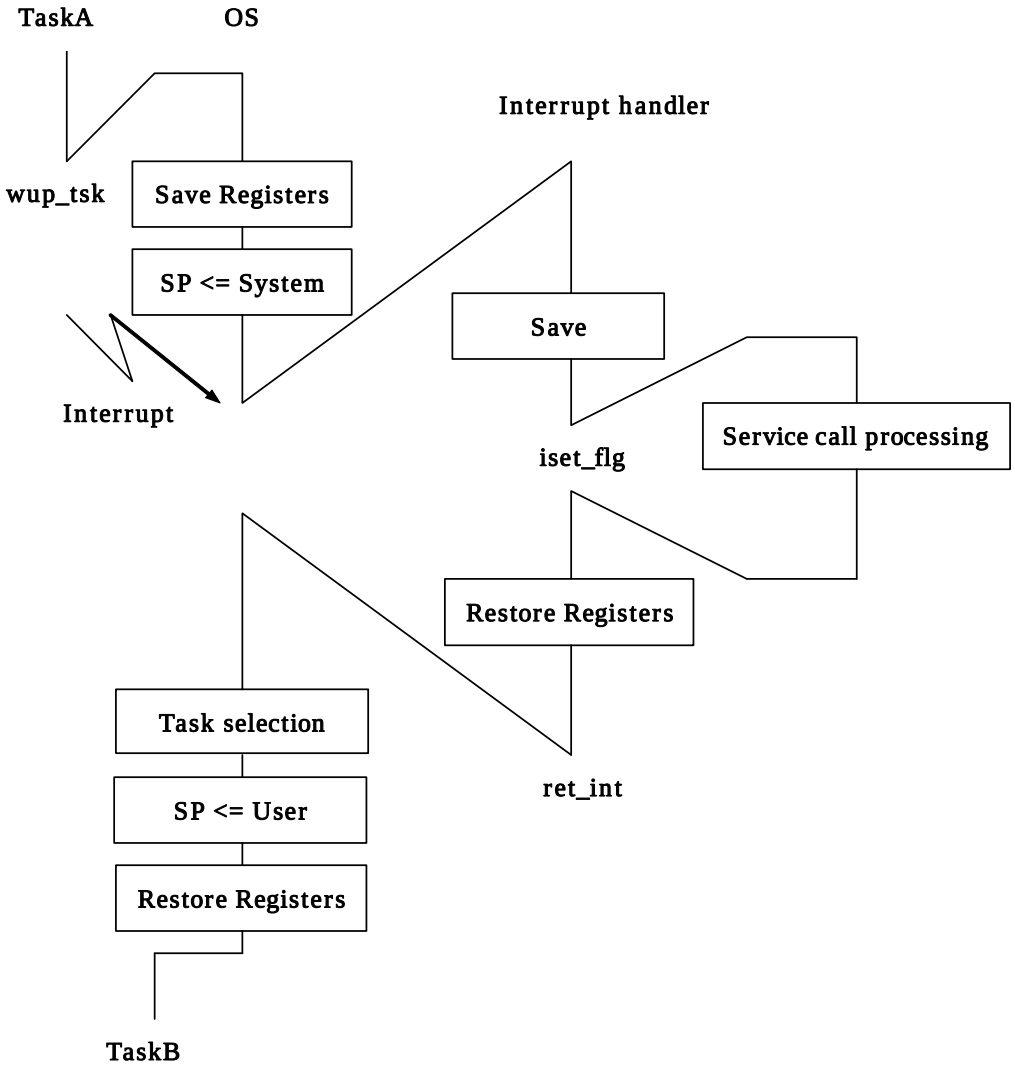


Figure 3.12 Processing Procedure for a Service Call from a Handler that caused an interrupt during Service Call Processing

Service Calls from a Handler That Caused an Interrupt during Handler Execution

Let us think of a situation in which an interrupt occurs during handler execution (this handler is hereinafter referred to as handler A for explanation purposes). When task switching is called for as a handler (hereinafter referred to as handler B) that caused an interrupt during handler A execution issued a service call, task switching does not take place during the execution of the service call (ret_int service call) returned from handler B, but is effected by the ret_int service call from handler A (See Figure 3.13).

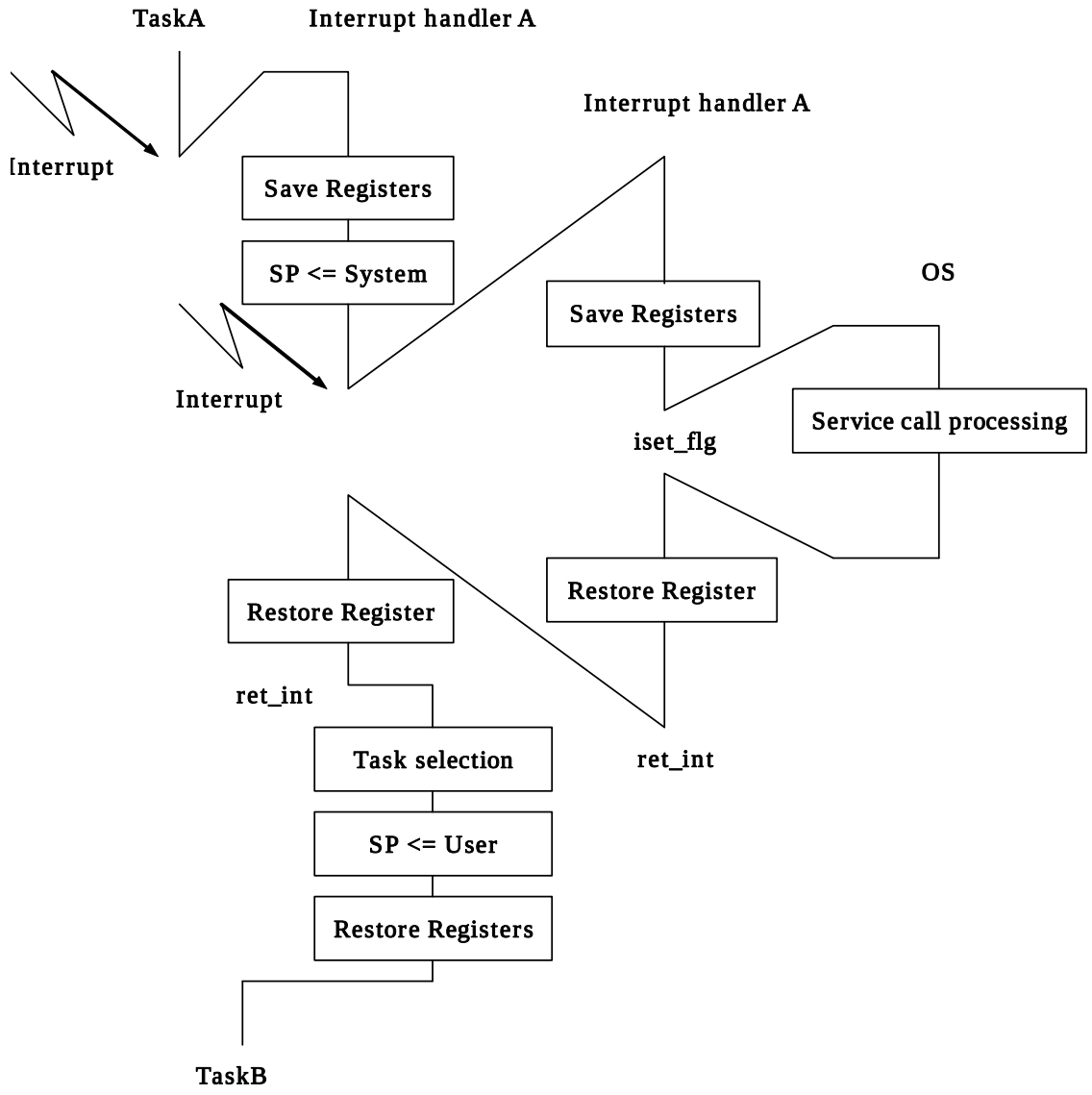


Figure 3.13 Processing Procedure for a service call from a Multiplex interrupt Handler

3.3 Object

The object operated by the service call of a semaphore, a task, etc. is called an "object." An object is identified by the ID number

3.3.1 The specification method of the object in a service call

Each task is identified by the ID number internally in MR8C/4.

For example, the system says, "Start the task having the task ID number 1."

However, if a task number is directly written in a program, the resultant program would be very low in readability. If, for instance, the following is entered in a program, the programmer is constantly required to know what the No. 2 task is.

```
sta_tsk(1, stacd);
```

Further, if this program is viewed by another person, he/she does not understand at a glance what the No. 1 task is. To avoid such inconvenience, the MR8C/4 provides means of specifying the task by name (function or symbol name).

The program named "configurator cfg8c ,"which is supplied with the MR8C/4, then automatically converts the task name to the task ID number. This task identification system is schematized in Figure 3.14.

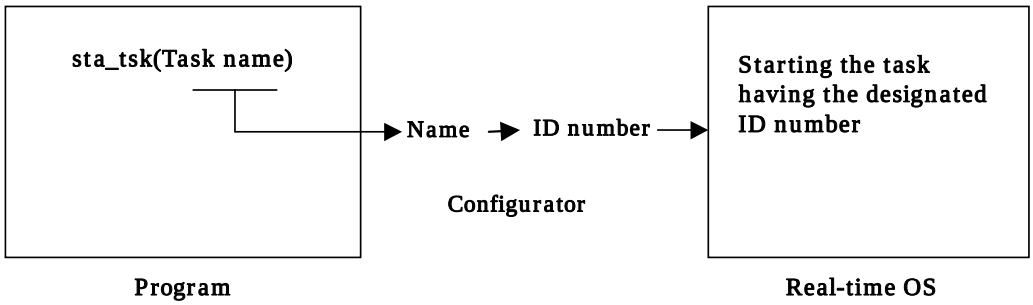


Figure 3.14 Task Identification

```
sta_tsk(ID_task, stacd);
```

This example specifies that a task corresponding to "ID_task" be invoked.

It should also be noted that task name-to-ID number conversion is effected at the time of program generation. Therefore, the processing speed does not decrease due to this conversion feature.

3.4 Task

This section describes how tasks are managed by MR8C/4.

3.4.1 Task Status

The real-time OS monitors the task status to determine whether or not to execute the tasks.

Figure 3.15 shows the relationship between key input task execution control and task status. When there is a key input, the key input task must be executed. That is, the key input task is placed in the execution (RUNNING) state. While the system waits for key input, task execution is not needed. In that situation, the key input task in the WAITING state.

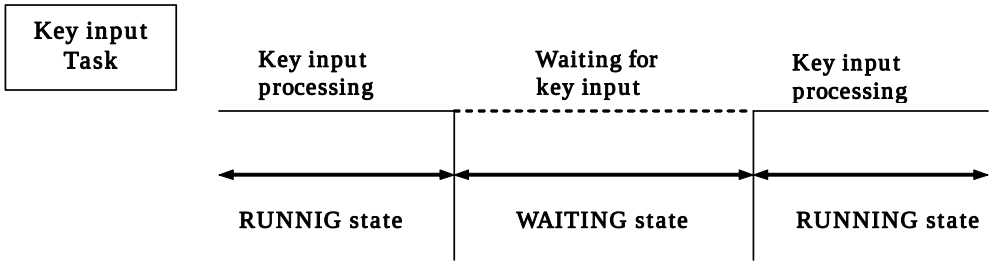


Figure 3.15 Task Status

The MR8C/4 controls the following six different states including the RUNNING and WAITING states.

1. **RUNNING state**
2. **READY state**
3. **WAITING state**
4. **SUSPENDED state**
5. **WAITING-SUSPENDED state**
6. **DORMANT state**

Every task is in one of the above six different states. Figure 3.16 shows task status transition.

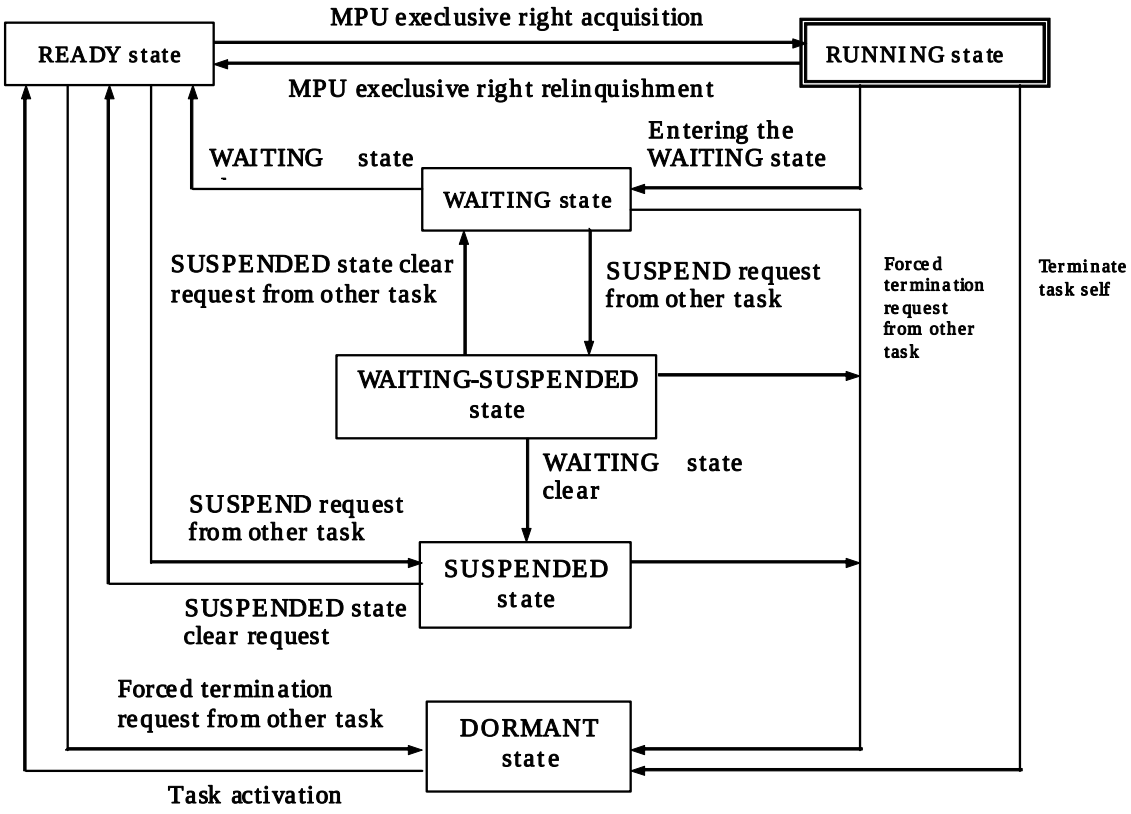


Figure 3.16 MR8C/4 Task Status Transition

1. RUNNING state

In this state, the task is being executed. Since only one microcomputer is used, it is natural that only one task is being executed.
The currently executed task changes into a different state when any of the following conditions occurs.

- The task has normally terminated itself by ext_tsk service call.
- The task has placed itself in the WAITING. [9]
- Since the service call was issued from the RUNNING state task, the WAITING state of another task with a priority higher than the RUNNING state task is cleared.
- Due to interruption or other event occurrence, the interrupt handler has placed a different task having a higher priority in the READY state.
- The priority assigned to the task has been changed by chg_pri service call so that the priority of another READY task is rendered higher.

When any of the above conditions occurs, rescheduling takes place so that the task having the highest priority among those in the RUNNING or READY state is placed in the RUNNING state, and the execution of that task starts.

2. READY state

The READY state refers to the situation in which the task that meets the task execution conditions is still waiting for execution because a different task having a higher priority is currently being executed.
When any of the following conditions occurs, the READY task that can be executed second according to the ready queue is placed in the RUNNING state.

- A currently executed task has normally terminated itself by ext_tsk service call.
- A currently executed task has placed itself in the WAITING state.[10]
- A currently executed task has changed its own priority by chg_pri service call so that the priority of

[9] By issuing dly_tsk, slp_tsk, wai_flg, wai_sem, snd_dtq and rcv_dtq service call.
[10] Depends on dly_tsk, slp_tsk, wai_flg, wai_sem, snd_dtq and rcv_dtq service call.

a different READY task is rendered higher.

- Due to interruption or other event occurrence, the priority of a currently executed task has been changed so that the priority of a different READY task is rendered higher.

3. WAITING state

When a task in the RUNNING state requests to be placed in the WAITING state, it exits the RUNNING state and enters the WAITING state. The WAITING state is usually used as the condition in which the completion of I/O device I/O operation or the processing of some other task is awaited.
The task goes into the WAITING state in one of the following ways.

- The task enters the WAITING state simply when the slp_tsk service call is issued. In this case, the task does not go into the READY state until its WAITING state is cleared explicitly by some other task.
- The task enters and remains in the WAITING state for a specified time period when the dly_tsk service call is issued. In this case, the task goes into the READY state when the specified time has elapsed or its WAITING state is cleared explicitly by some other task.
- The task is placed into WAITING state for a wait request by the wai_flg, wai_sem,snd_dtq or rcv_dtq service call. In this case, the task goes from WAITING state to READY state when the request is met or WAITING state is explicitly canceled by another task.
- If the task is placed into WAITING state for a wait request by the wai_flg, wai_sem, snd_dtq, or rcv_dtq service call, the task is queued to one of the following waiting queues depending on the request.

    - Event flag waiting queue
    - Semaphore waiting queue
    - Data queue data transmission waiting queue
    - Data queue data reception waiting queue

4. SUSPENDED state

When the sus_tsk service call is issued from a task in the RUNNING state , the READY task designated by the service call or the currently executed task enters the SUSPENDED state. If a task in the WAITING state is placed in this situation, it goes into the WAITING-SUSPENDED state.

The SUSPENDED state is the condition in which a READY task or currently executed task is excluded from scheduling to halt processing due to I/O or other error occurrence. That is, when the suspend request is made to a READY task, that task is excluded from the execution queue.

Note that no queue is formed for the suspend request. Therefore, the suspend request can only be made to the tasks in the RUNNING, READY, or WAITING state.[11] If the suspend request is made to a task in the SUSPENDED state, an error code is returned.

5. WAITING-SUSPENDED

If a suspend request is issued to a task currently in a WAITING state, the task goes to a WAITING-SUSPENDED state. If a suspend request is issued to a task that has been placed into a WAITING state for a wait request by the slp_tsk, wai_flg, wai_sem, snd_dtq or rcv_dtq service call, the task goes to a WAITING-SUSPENDED state.

When the wait condition for a task in the WAITING-SUSPENDED state is cleared, that task goes into the SUSPENDED state. It is conceivable that the wait condition may be cleared, when any of the following conditions occurs.

- The task wakes up upon wup_tsk, or iwup_tsk service call issuance.
- The task placed in the WAITING state by the dly_tsk service call wakes up after the specified time elapse.
- The request of the task placed in the WAITING state by the wai_flg , wai_sem, snd_dtq or rcv_dtq service call is fulfilled.
- The WAITING state is forcibly cleared by the rel_wai or irel_wai service call

[11] If a forcible wait request is issued to a task currently in a wait state, the task goes to a WAITING-SUSPENDED state.

When the SUSPENDED state clear request by rsm_tsk or irsm_tsk is made to a task in the WAITING-SUSPENDED state, that task goes into the WAITING state. Since a task in the SUSPENDED state cannot request to be placed in the WAITING state, status change from SUSPENDED to WAITING-SUSPENDED does not possibly occur.

6. DORMANT

This state refers to the condition in which a task is registered in the MR8C/4 system but not activated. This task state prevails when either of the following two conditions occurs.

- The task is waiting to be activated.
- The task is normally terminated by ext_tsk service call or forcibly terminated by ter_tsk service call.

3.4.2 Task Priority and Ready Queue

In the kernel, several tasks may simultaneously request to be executed. In such a case, it is necessary to determine which task the system should execute first. To properly handle this kind of situation, the system organizes the tasks into proper execution priority and starts execution with a task having the highest priority. To complete task execution quickly, tasks related to processing operations that need to be performed immediately should be given higher priorities.

The MR8C/4 permits giving the same priority to several tasks. To provide proper control over the READY task execution order, the kernel generates a task execution queue called "ready queue." The ready queue structure is shown in Figure 3.17[12] The ready queue is provided and controlled for each priority level. The first task in the ready queue having the highest priority is placed in the RUNNING state.[13]

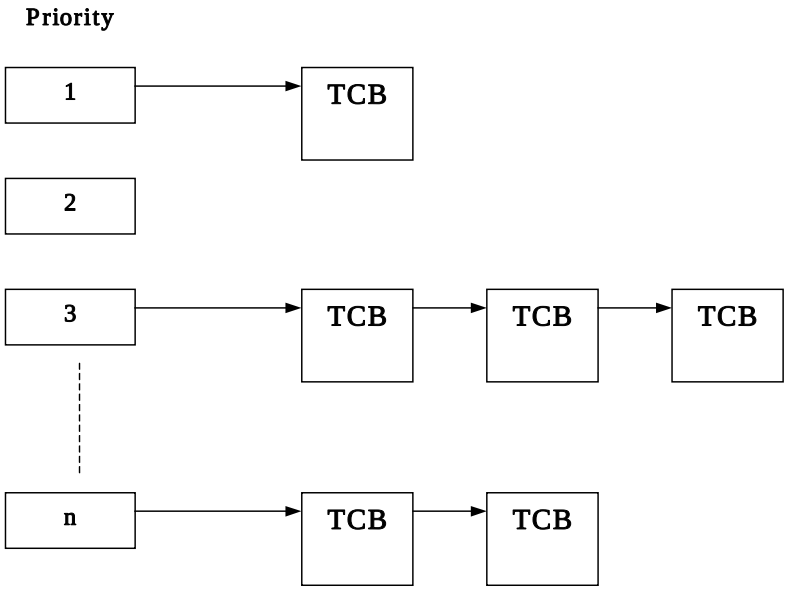


Figure 3.17 Ready Queue (Execution Queue)

[12] The TCB(task control block is described in the next chapter.)
[13] The task in the RUNNING state remains in the ready queue.

3.4.3 Task Priority and Waiting Queue

In The standard profiles in µITRON 4.0 Specification support two waiting methods for each object. In one method, tasks are placed in a waiting queue in order of priority (TA_TPRI attribute); in another, tasks are placed in a waiting queue in order of FIFO (TA_TFIFO).

MR8C/4 supports only TA_TFIFO attribute.

Figure 3.19 depict the manner in which tasks are placed in a waiting queue in order of "taskD," "taskC," "taskA," and "taskB."

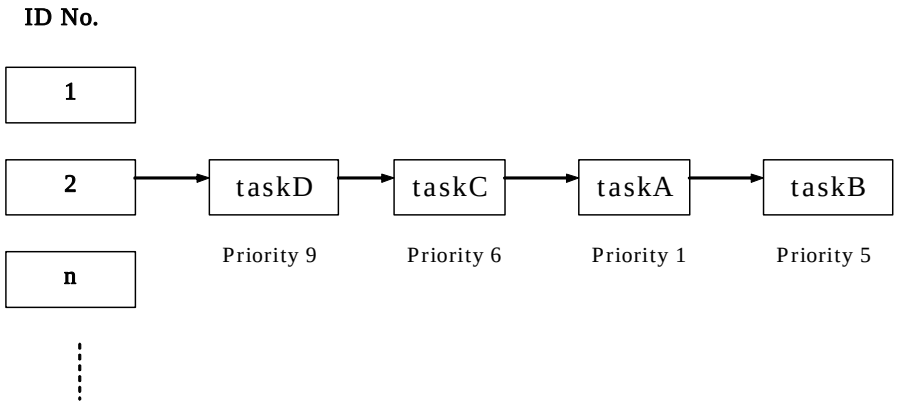


Figure 3.18 Waiting queue of the TA_TFIFO attribute

3.4.4 Task Control Block(TCB)

The task control block (TCB) refers to the data block that the real-time OS uses for individual task status, priority, and other control purposes.

The MR8C/4 manages the following task information as the task control block

- Task connection pointer
  Task connection pointer used for ready queue formation or other purposes.
- Task status
- Task priority
- Task register information and other data[14] storage stack area pointer(current SP register value)
- Wake-up counter
  Task wake-up request storage area.
- Flag wait pattern
  This area is used when using the timeout function.
  This area stores the flag wait pattern when using the eventflag wait service call with the timeout function. No flag wait pattern area is allocated when the eventflag is not used.
- Flag wait mode
  This is a wait mode during eventflag wait.
- Delay time counter
  Delay time counter when dly_tsk is called..
- Extended task information
  Extended task information that was set during task generation is stored in this area.

[14] Called the task context

The task control block is schematized in Figure 3.19.

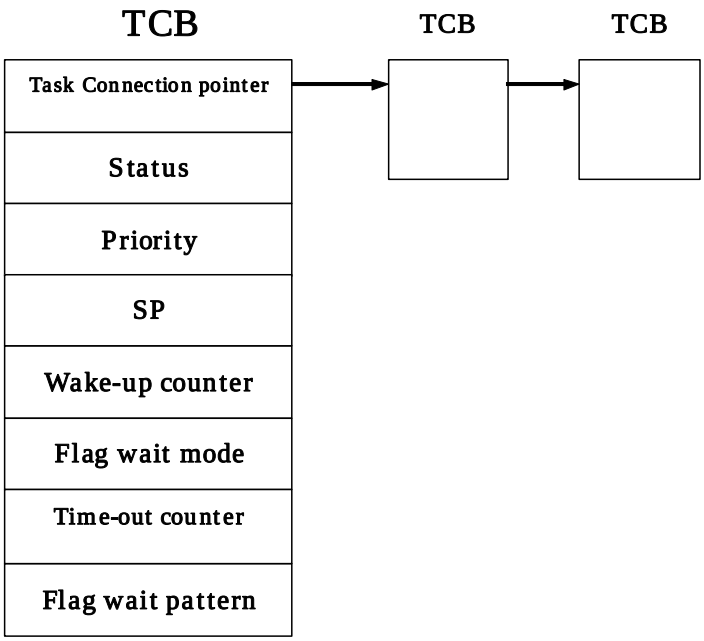


Figure 3.19 Task control block

3.5 System States

3.5.1 Task Context and Non-task Context

The system runs in either context state, "task context" or "non-task context." The differences between the task content and non-task context are shown in Table 3-1. Task Context and Non-task Context.

Table 3.1 Task Context and Non-task Context

	Task context	Non-task context
Invocable service call	Those that can be invoked from task context	Those that can be invoked from non-task context
Task scheduling	Occurs when the queue state has changed to other than dispatch disabled and CPU locked states.	It does not occur.
Stack	User stack	System stack

The processes executed in non-task context include the following.

1. Interrupt Handler

A program that starts upon hardware interruption is called the interrupt handler. The MR8C/4 is not concerned in interrupt handler activation. Therefore, the interrupt handler entry address is to be directly written into the interrupt vector table.

There are two interrupt handlers: Non-kernel interrupts (OS independent interrupts) and kernel interrupts (OS dependent interrupts). For details about each type of interrupt, refer to Section 3.6.

The system clock interrupt handler (isig_tim) is one of these interrupt handlers.

2. Cyclic Handler

The cyclic handler is a program that is started cyclically every preset time. The set cyclic handler may be started or stopped by the sta_cyc or stp_cyc service call.

3. Alarm Handler

The alarm handler is a handler that is started after the lapse of a specified relative time of day. The alarm handler startup time of day is determined by a time of day relative to the time of day set by sta_alm.

The cyclic and alarm handlers are invoked by a subroutine call from the system clock interrupt (timer interrupt) handler. Therefore, cyclic and alarm handlers operate as part of the system clock interrupt handler. Note that when the cyclic or alarm handler is invoked, it is executed in the interrupt priority level of the system clock interrupt.

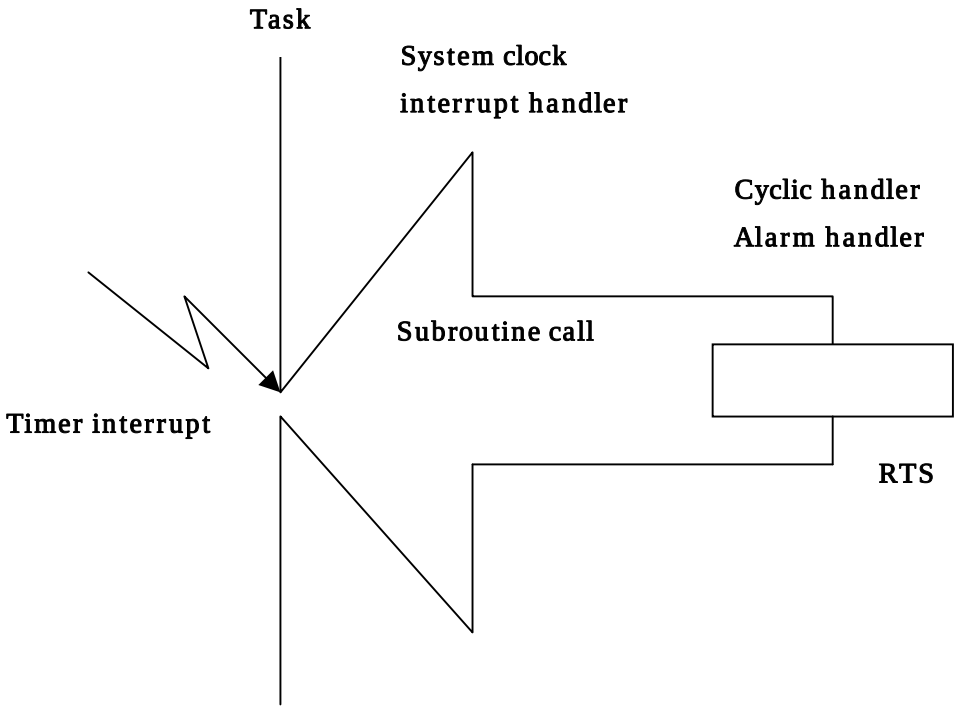


Figure 3.20 Cyclic Handler/Alarm Handler Activation

3.5.2 Dispatch Enabled/Disabled States

The system assumes either a dispatch enabled state or a dispatch disabled state. In a dispatch disabled state, no task scheduling is performed. Nor can service calls be invoked that may cause the service call issuing task to enter a wait state.[15]

The system can be placed into a dispatch disabled state or a dispatch enabled state by the dis_dsp or ena_dsp service call, respectively. Whether the system is in a dispatch disabled state can be known by the sns_dsp service call.

3.5.3 CPU Locked/Unlocked States

The system assumes either a CPU locked state or a CPU unlocked state. In a CPU locked state, all external interrupts are disabled against acceptance, and task scheduling is not performed either.

The system can be placed into a CPU locked state or a CPU unlocked state by the loc_cpu(iloc_cpu) or unl_cpu(iunl_cpu) service call, respectively. Whether the system is in a CPU locked state can be known by the sns_loc service call.

The service calls that can be issued from a CPU locked state are limited to those that are listed in Table 3-2.[16]

Table 3.2 Invocable Service Calls in a CPU Locked State

loc_cpu	unl_cpu	ext_tsk	sns_dsp
sns_loc	sns_ctx		

3.5.4 Dispatch Disabled and CPU Locked States

In µITRON 4.0 Specification, the dispatch disabled and the CPU locked states are clearly discriminated. Therefore, if the unl_cpu service call is issued in a dispatch disabled state, the dispatch disabled state remains intact and no task scheduling is performed. State transitions are summarized in Table 3.3.

Table 3.3 CPU Locked and Dispatch Disabled State Transitions Relating to dis_dsp and loc_cpu

State number	Content of state		dis_dsp executed	ena_dsp executed	loc_cpu executed	unl_cpu executed
	CPU locked state	Dispatch disabled state				
1	O	X	X	X	=> 1	=> 3
2	O	O	X	X	=> 2	=> 4
3	X	X	=> 4	=> 3	=> 1	=> 3
4	X	O	=> 4	=> 3	=> 2	=> 4

[15] If a service call not issuable is issued when dispatch disabled, MR8C/4 doesn't return the error and doesn't guarantee the operation.
[16] MR8C/4 does not return an error even when an uninvocable service call is issued from a CPU locked state, in which case, however, its operation cannot be guaranteed.

3.6 Regarding Interrupts

3.6.1 Types of Interrupt Handlers

MR8C/4's interrupt handlers consist of kernel(OS-dependent) interrupt handlers and non-kernel (OS-independent) interrupt handlers.

The following shows the definition of each type of interrupt handler.

- Kernel(OS-dependent) interrupt handler
  An interrupt handler whose interrupt priority level is lower than a kernel interruption mask level (OS interrupt prohibition level) is called kernel (OS dependent) interrupt handler. That is, interruption priority level is from 0 to system_IPL.
  A service call can be issued within a kernel (OS dependent) interrupt handler. However, interrupt is delayed until it becomes receivable [the kernel management (OS dependence) interrupt handler generated during service call processing / kernel management (OS dependence) interruption].

- Non-kernel(OS-independent) interrupt handler
  An interrupt handler whose interrupt priority level is higher than a kernel interrupt mask level (OS interrupt prohibition level)is called non-kernel interrupt handler (OS independent handler) That is, interruption priority level is from system_IPL+1 to 7.
  A service call cannot be published within an interruption (OS independence)-kernel management outside hair drier. However, the kernel management generated during service call processing outside, even if it is the section where interruption cannot receive a kernel management (OS dependence) interrupt handler (OS independence), it is possible to receive interruption kernel management outside (OS independence).:

Figure 3.21 shows the relationship between the non-kernel(OS-independent) interrupt handlers and kernel(OS-dependent) interrupt handlers where the kernel mask level(OS interrupt disable level) is set to 3.

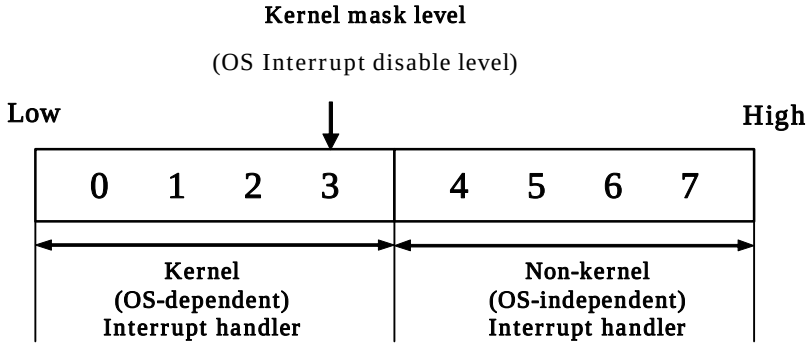


Figure 3.21 Interrupt handler IPLs

3.6.2 The Use of Non-maskable Interrupt

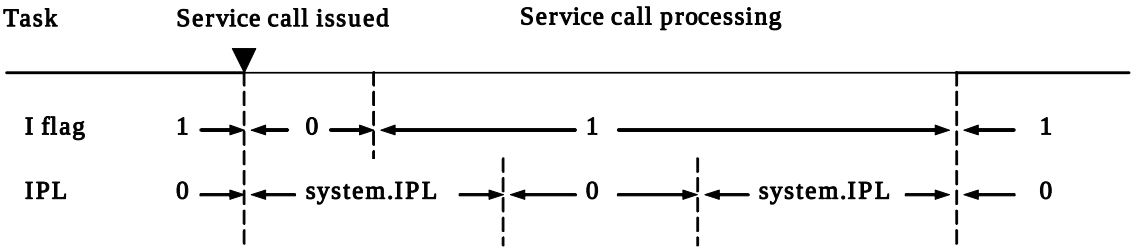
An NMI interrupt and Watchdog Timer interrupt have to use be a non-kernel(OS independent) interrupt handler. If they are a kernel(OS dependent) interrupt handler, the program will not work normally.

3.6.3 Controlling Interrupts

Interrupt enable/disable control in a service call is accomplished by IPL manipulation. The IPL value in a service call is set to the kernel mask level(OS interrupt disable level = system.IPL) in order to disable interrupts for the kernel (OS-dependent) interrupt handler. In sections where all interrupts can be enabled, it is returned to the initial IPL value when the service call was invoked.

- For service calls that can be issued from only task context.

  When the I flag before issuing a service call is 1.



  When the I flag before issuing a service call is 0.

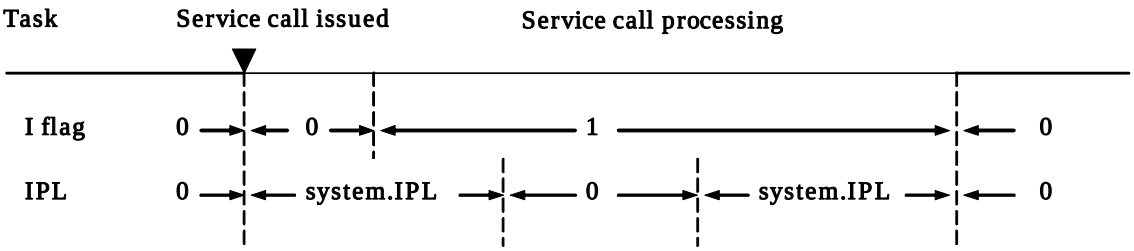
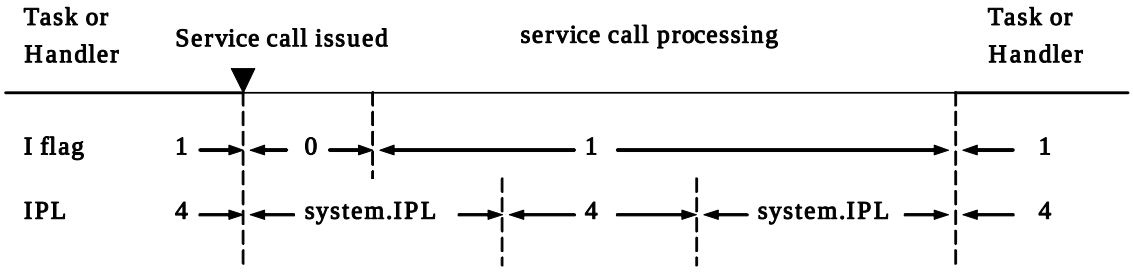


Figure 3.22 Interrupt control in a Service Call that can be Issued from only a Task

- For service calls that can be issued from only non-task context or from both task context and non-task context.

When the I flag before issuing a service call is 1



When the I flag before issuing a service call is 0

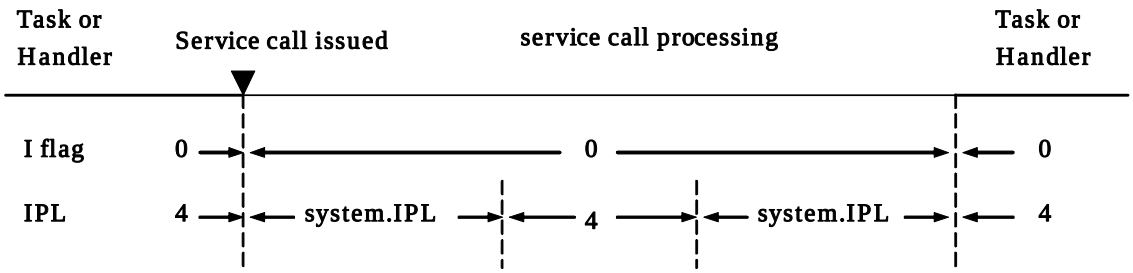


Figure 3.23 Interrupt control in a Service Call that can be Issued from a Task-independent

As shown in Figure 3.22 and Figure 3.23, the interrupt enable flag and IPL change in a service call. For this reason, if you want to disable interrupts in a user application, Renesas does not recommend using the method for manipulating the interrupt disable flag and IPL to disable the interrupts.

The following two methods for interrupt control are recommended:

1. **Modify the interrupt control register (SFR) for the interrupt you want to be disabled.**
2. **Use service calls loc_cpu and unl_cpu.**

The interrupts that can be controlled by the loc_cpu service call are only the kernel(OS-dependent) interrupt. Use method 1 to control the non-kernel(OS-independent) interrupts.

3.7 Stacks

3.7.1 System Stack and User Stack

The MR8C/4 provides two types of stacks: system stack and user stack.

- User Stack
  One user stack is provided for each task. Therefore, when writing applications with the MR8C/4, it is necessary to furnish the stack area for each task.

- System Stack
  This stack is used within the MR8C/4 (during service call processing). When a service call is issued from a task, the MR8C/4 switches the stack from the user stack to the system stack (See Figure 3.24).
  The system stack use the interrupt stack(ISP).

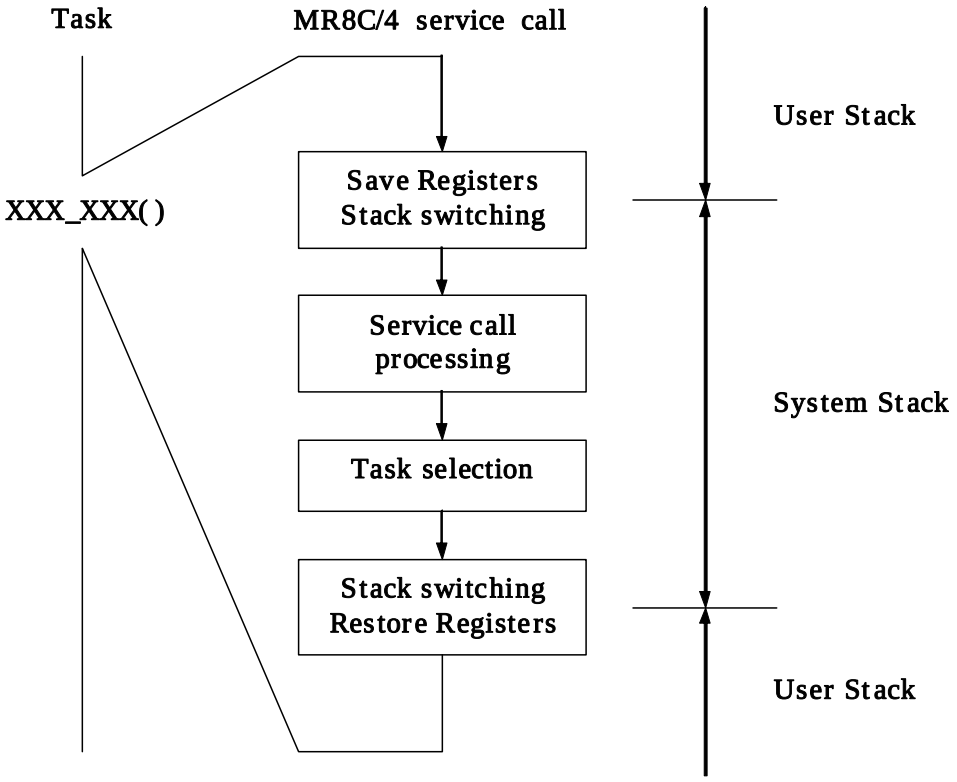


Figure 3.24 System Stack and User Stack

Switchover from user stack to system stack occurs when an interrupt of vector numbers 0 to 31 or 247 to 255 is generated. Consequently, all stacks used by the interrupt handler are the system stack.

4. Kernel

4.1 Module Structure

The MR8C/4 kernel consists of the modules shown in Figure 4.1. Each of these modules is composed of functions that exercise individual module features.

The MR8C/4 kernel is supplied in the form of a library, and only necessary features are linked at the time of system generation. More specifically, only the functions used are chosen from those which comprise these modules and linked by means of the Linkage Editor LN30. However, the scheduler module, part of the task management module, and part of the time management module are linked at all times because they are essential feature functions.

The applications program is a program created by the user. It consists of tasks, interrupt handler, alarm handler, and cyclic handler.

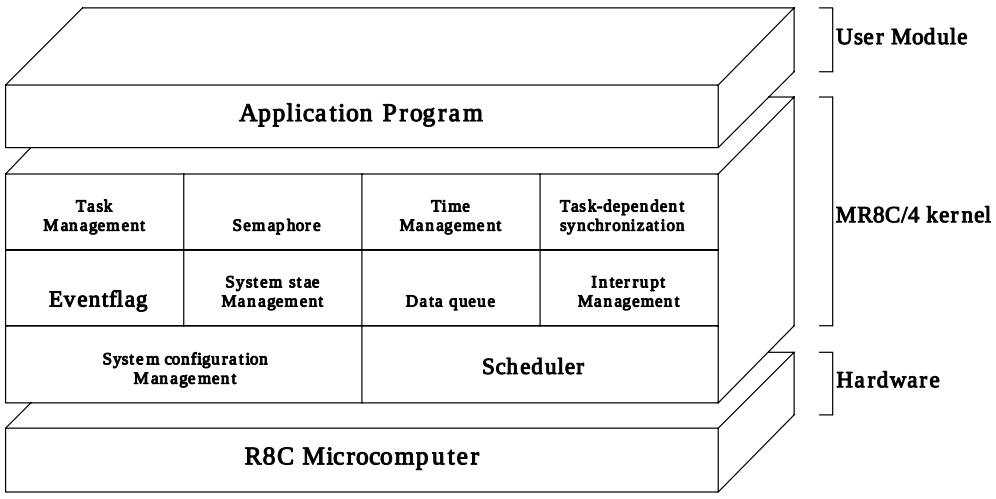


Figure 4.1 MR8C/4 Structure

4.2 Module Overview

The MR8C/4 kernel modules are outlined below.

- **Scheduler**
  Forms a task processing queue based on task priority and controls operation so that the high-priority task at the beginning in that queue (task with small priority value) is executed.
- **Task Management Module**
  Exercises the management of various task states such as the RUNNING, READY, WAIT, and SUSPENDED state.
- **Task Dependent Synchronization Module**
  Accomplishes inter-task synchronization by changing the task status from a different task.
- **Interrupt Management Module**
  Makes a return from the interrupt handler.
- **Time Management Module**
  Sets up the system timer used by the MR8C/4 kernel and starts the user-created alarm handler[17] and cyclic handler.[18].
- **System Status Management Module**
  Gets the system status of MR8C/4.
- **System Configuration Management Module**
  Reports the MR8C/4 kernel version number or other information.
- **Sync/Communication Module**
  This is the function for synchronization and communication among the tasks. The following three functional modules are offered.

  - Eventflag
    Checks whether the flag controlled within the MR8C/4 is set up and then determines whether or not to initiate task execution. This results in accomplishing synchronization between tasks.
  - Semaphore
    Reads the semaphore counter value controlled within the MR8C/4 and then determines whether or not to initiate task execution. This also results in accomplishing synchronization between tasks.
  - Data queue
    Performs 16-bit data communication between tasks.

[17] This handler actuates once only at preselected times.
[18] This handler periodically actuates.

4.2.1 Task Management Function

The task management function is used to perform task operations such as task start/stop and task priority updating. The MR8C/4 kernel offers the following task management function service calls.

- Activate Task (sta_tsk, ista_tsk)
  Activates the task, changing its status from DORMANT to either READY or RUNNING. In this service call, unlike in act_tsk(iact_tsk), startup requests are not accumulated, but startup code can be specified.

- Terminate Invoking Task (ext_tsk)
  When the issuing task is terminated, its state changes to DORMANT state. The task is therefore not executed until it is restarted. If startup requests are accumulated, task startup processing is performed again. In that case, the issuing task behaves as if it were reset.
  If written in C language, this service call is automatically invoked at return from the task regardless of whether it is explicitly written when terminated.

- Terminate Task (ter_tsk)
  Other tasks in other than DORMANT state are forcibly terminated and placed into DORMANT state. If startup requests are accumulated, task startup processing is performed again. In that case, the task behaves as if it was reset. (See Figure 4.2).

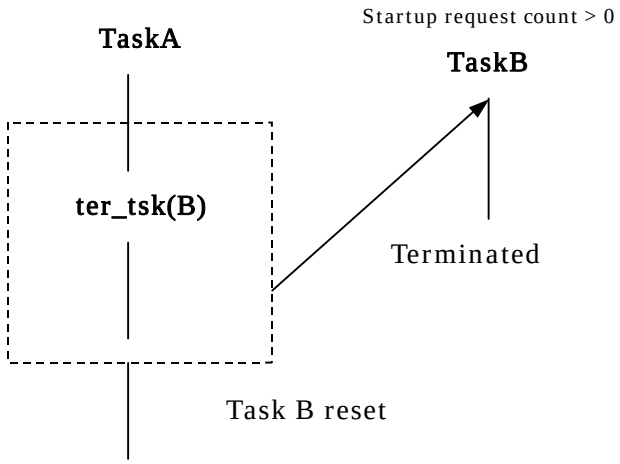


Figure 4.2 Task Resetting

- Change Task Priority (chg_pri)
  If the priority of a task is changed while the task is in READY or RUNNING state, the ready queue also is updated. (See Figure 4.3).

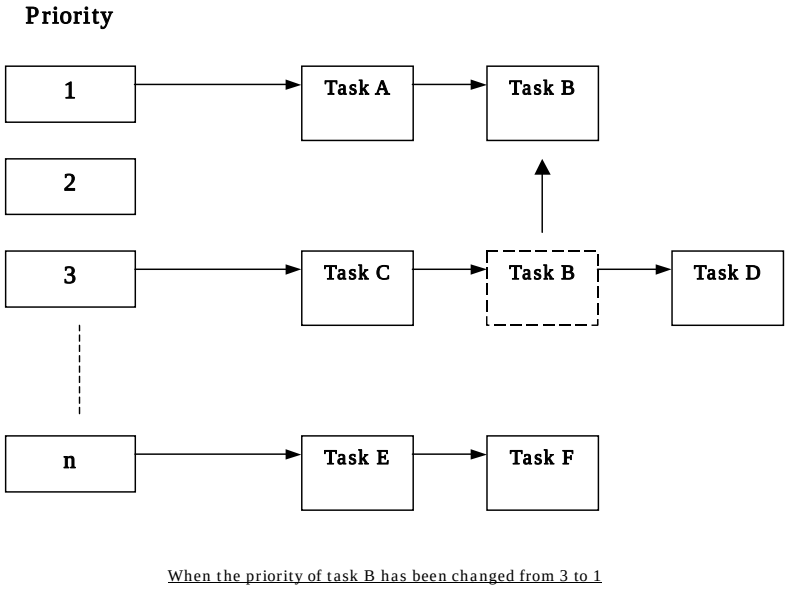


Figure 4.3 Alteration of task priority

- Reference task status (simple version) (ref_tst, iref_tst)
  Refers to the state of the target task.

- Reference task status (ref_tsk, iref_tsk)
  Refers to the state of the target task and its priority, etc.

4.2.2 Task Dependent Synchronization Function

The task-dependent synchronization functions attached to task is used to accomplish synchronization between tasks by placing a task in the WAIT, SUSPENDED, or WAIT-SUSPENDED state or waking up a WAIT state task.

The MR8C/4 offers the following task incorporated synchronization service calls.

- Put Task to sleep (slp_tsk)
- Wakeup task (wup_tsk, iwup_tsk)
  Wakeups a task that has been placed in a WAITING state by the slp_tsk service call.
  No task can be waked up unless they have been placed in a WAITING state by.
  If a wakeup request is issued to a task that has been kept waiting for conditions other than the slp_tsk service call or a task in other than DORMANT state by the wup_tsk or iwup_tsk service call, that wakeup request only will be accumulated.
  Therefore, if a wakeup request is issued to a task RUNNING state, for example, this wakeup request is temporarily stored in memory. Then, when the task in RUNNING state is going to be placed into WAIT state by the slp_tsk service call, the accumulated wakeup request becomes effective, so that the task continues executing again without going to WAIT state. (See Figure 4.4).
- Cancel Task Wakeup Requests (can_wup)
  Clears the stored wakeup request.(See Figure 4.5).

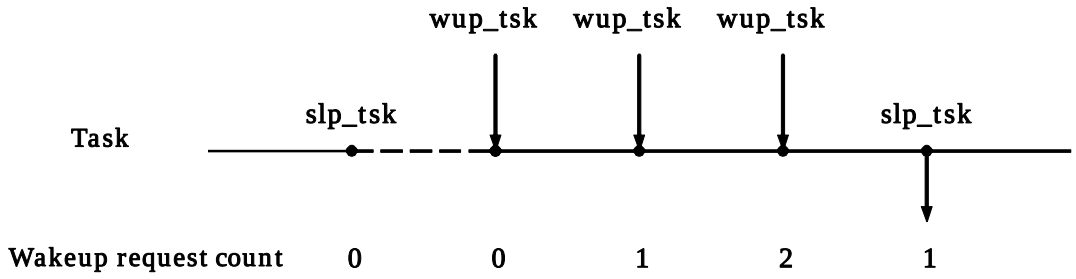


Figure 4.4 Wakeup Request Storage

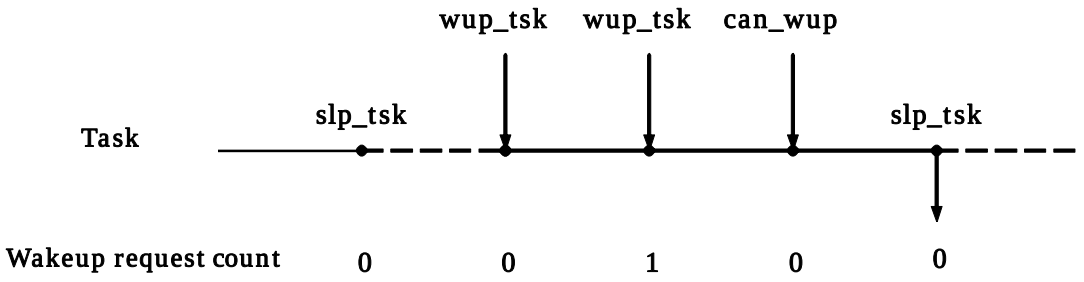


Figure 4.5 Wakeup Request Cancellation

- Suspend task (sus_tsk)
- Resume suspended task (rsm_tsk)

  These service calls forcibly keep a task suspended for execution or resume execution of a task. If a suspend request is issued to a task in READY state, the task is placed into SUSPENDED state; if issued to a task in WAIT state, the task is placed into WAIT-SUSPENDED state. Since MR8C/4 allows only one forcible wait request to be nested, if sus_tsk is issued to a task in a forcible wait state, the error E_QOVR is returned. (See Figure 4.6).

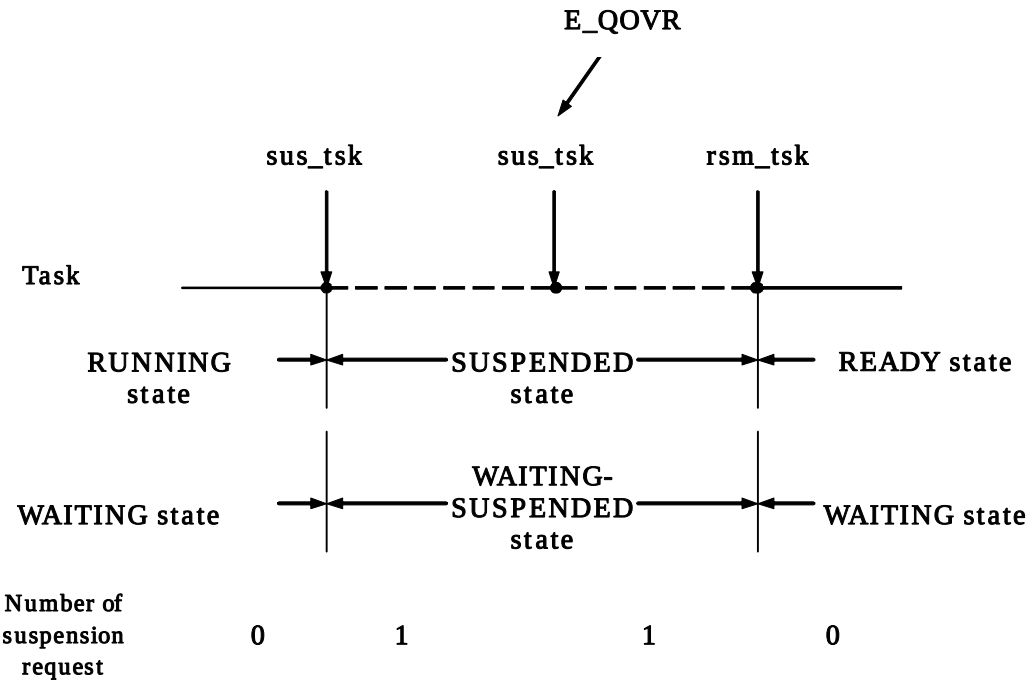


Figure 4.6 Forcible wait of a task and resume

- Release task from waiting (rel_wai, irel_wai)
  Forcibly frees a task from WAITING state. A task is freed from WAITING state by this service call when it is in one of the following wait states.

  - Delay wait state
  - Wait state entered by slp_tsk service call
  - Event flag wait state
  - Semaphore wait state
  - Data transmission wait state
  - Data reception wait state

- Delay task (dly_tsk)
  Keeps a task waiting for a finite length of time. Figure 4.7 shows an example in which execution of a task is kept waiting for 10 ms by the dly_tsk service call. The delay time value should be specified in ms units, and not in time tick units.

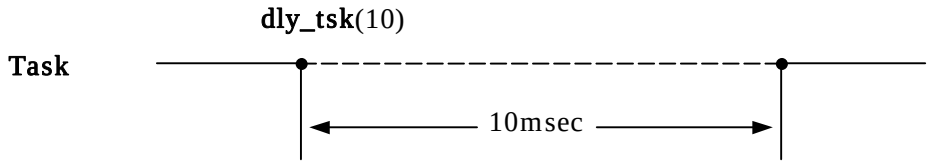


Figure 4.7 dly_tsk service call

4.2.3 Synchronization and Communication Function (Semaphore)

The semaphore is a function executed to coordinate the use of devices and other resources to be shared by several tasks in cases where the tasks simultaneously require the use of them. When, for instance, four tasks simultaneously try to acquire a total of only three communication lines as shown in Figure 4.8, communication line-to-task connections can be made without incurring contention.

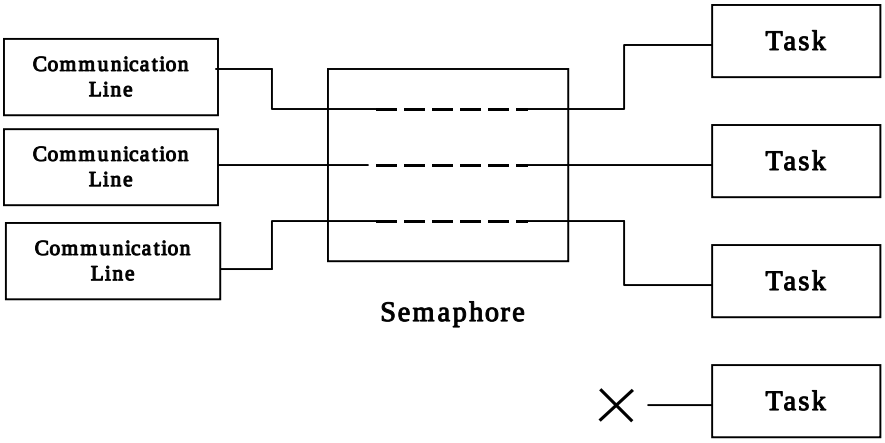


Figure 4.8 Exclusive Control by Semaphore

The semaphore has an internal semaphore counter. In accordance with this counter, the semaphore is acquired or released to prevent competition for use of the same resource.(See Figure 4.9).

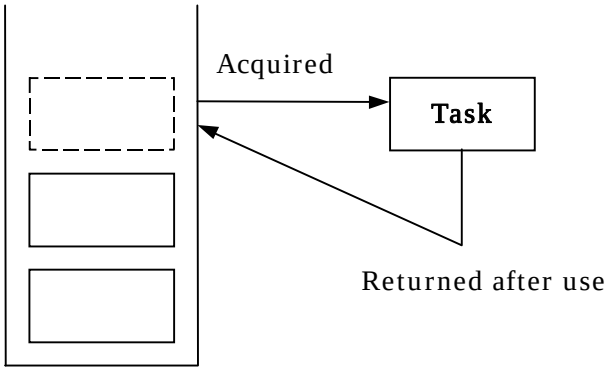


Figure 4.9 Semaphore Counter

Figure 4.10 shows example task execution control provided by the wai_sem and sig_sem service calls.

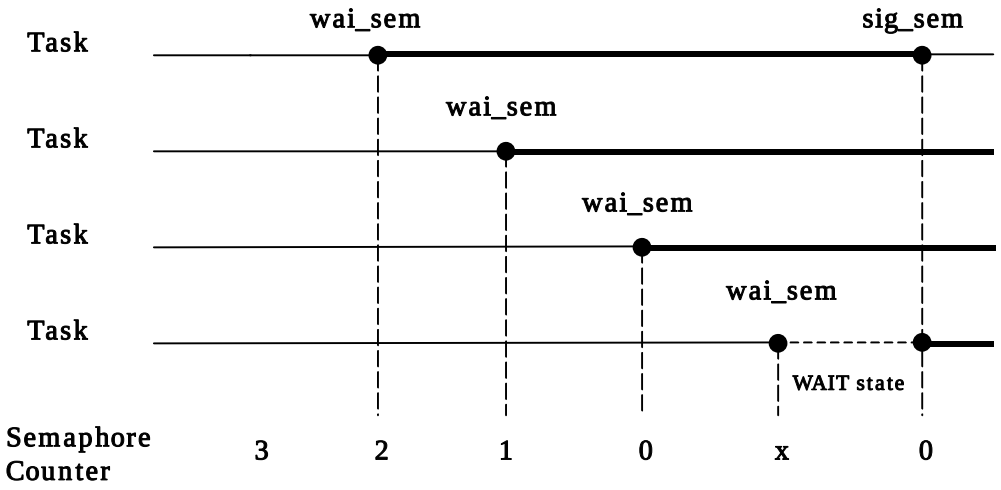


Figure 4.10 Task Execution Control by Semaphore

The MR8C/4 kernel offers the following semaphore synchronization service calls.

- Release Semaphore Resource(sig_sem, isig_sem)
  Releases one resource to the semaphore. This service call wakes up a task that is waiting for the semaphores service, or increments the semaphore counter by 1 if no task is waiting for the semaphores service.

- Acquire Semaphore Resource(wai_sem)
  Waits for the semaphores service. If the semaphore counter value is 0 (zero), the semaphore cannot be acquired. Therefore, the WAITING state prevails.

- Acquire Semaphore Resource(pol_sem)
  Acquires the semaphore resource. If there is no semaphore resource to acquire, an error code is returned and the WAITING state does not prevail.

4.2.4 Synchronization and Communication Function (Eventflag)

The eventflag is an internal facility of MR8C/4 that is used to synchronize the execution of multiple tasks. The eventflag uses a flag wait pattern and a 16-bit pattern to control task execution. A task is kept waiting until the flag wait conditions set are met.

It is possible to determine whether multiple waiting tasks can be enqueued in one eventflag waiting queue by specifying the eventflag attribute TA_WSGL or TA_WMUL.

Furthermore, it is possible to clear the eventflag bit pattern to 0 when the eventflag meets wait conditions by specifying TA_CLR for the eventflag attribute.

Figure 4.11 shows an example of task execution control by the eventflag using the wai_flg and set_flg service calls.

The eventflag has a feature that it can wake up multiple tasks collectively at a time.

In Figure 4.11, there are six tasks linked one to another, task A to task F. When the flag pattern is set to 0xF by the set_flg service call, the tasks that meet the wait conditions are removed sequentially from the top of the queue. In this diagram, the tasks that meet the wait conditions are task A, task C, and task E. Out of these tasks, task A, task C, and task E are removed from the queue.

If this event flag has a TA_CLR attribute, when the waiting of Task A is canceled, the bit pattern of the event flag will be set to 0, and Task C and Task E will not be removed from queue.

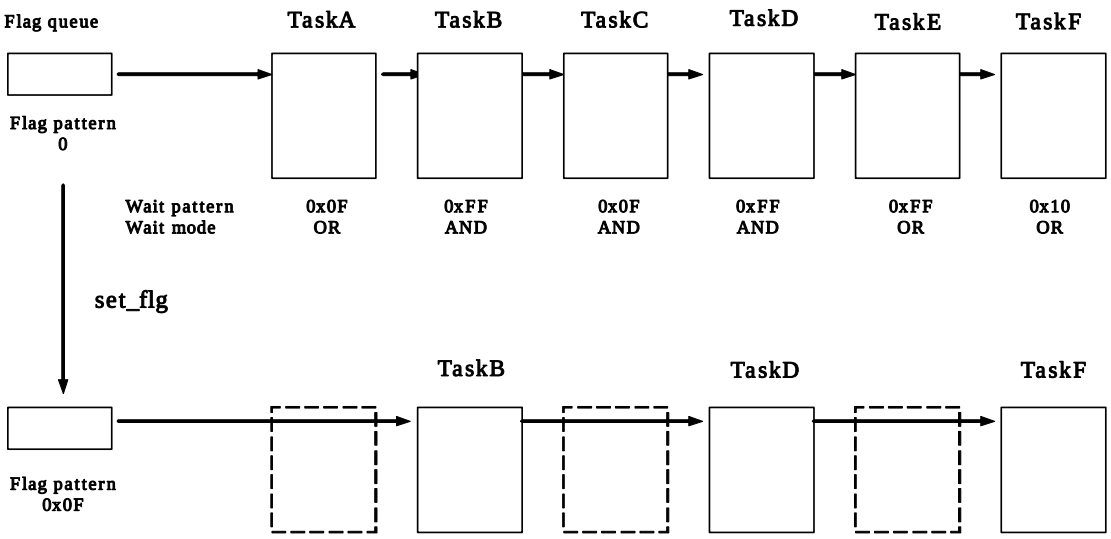


Figure 4.11 Task Execution Control by the Eventflag

There are following eventflag service calls that are provided by the MR8C/4 kernel.

- Set Eventflag (set_flg, iset_flg)
  Sets the eventflag so that a task waiting the eventflag is released from the WAITING state.

- Clear Eventflag (clr_flg)
  Clears the Eventflag.

- Wait for Eventflag (wai_flg)
  Waits until the eventflag is set to a certain pattern. There are two modes as listed below in which the eventflag is waited for.

  - AND wait
    Waits until all specified bits are set.

  - OR wait
    Waits until any one of the specified bits is set

- Wait for Eventflag (polling)(pol_flg)
  Examines whether the eventflag is in a certain pattern. In this service call, tasks are not placed in WAITING state.

4.2.5 Synchronization and Communication Function (Data Queue)

The data queue is a mechanism to perform data communication between tasks. In Figure 4.12, for example, task A can transmit data to the data queue and task B can receive the transmitted data from the data queue.

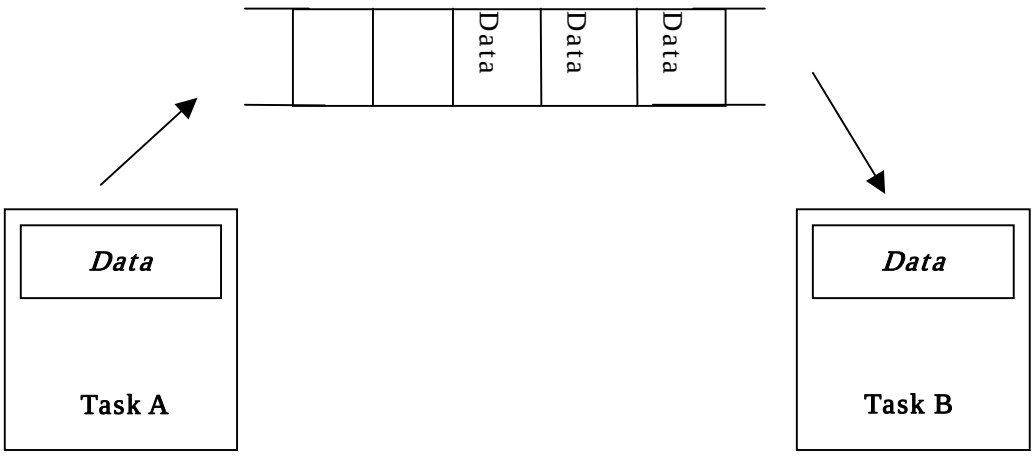


Figure 4.12 Data queue

Data in width of 16 bits can be transmitted to this data queue.

The data queue has the function to accumulate data. The accumulated data is retrieved in order of FIFO[19]. However, the number of data that can be accumulated in the data queue is limited. If data is transmitted to the data queue that is full of data, the service call issuing task goes to a data transmission wait state.

There are following data queue service calls that are provided by the MR8C/4 kernel.

- Send to Data Queue(snd_dtq)
  The data is transmitted to the data queue. If the data queue is full of data, the task goes to a data transmission wait state.
- Send to Data Queue (psnd_dtq, ipsnd_dtq)
  The data is transmitted to the data queue. If the data queue is full of data, the task returns error code without going to a data transmission wait state.
- Receive from Data Queue (rcv_dtq)
  The data is retrieved from the data queue. If the data queue has no data in it, the task is kept waiting until data is transmitted to the data queue.
- Receive from Data Queue (prcv_dtq)
  The data is received from the data queue. If the data queue has no data in it, the task returns error code without going to a data reception wait state.

[19] First In First Out

4.2.6 Time Management Function

The time management function provides system time management, and the functions of the alarm handler, which actuates at preselected times, and the cyclic handler, which actuates at preselected time intervals.

The MR8C/4 kernel requires one timer for use as the system clock. There are following time management service calls that are provided by the MR8C/4 kernel. Note, however, that the system clock is not an essential function of MR8C/4. Therefore, if the service calls described below and the time management function of the MR8C/4 are unused, a timer does not need to be occupied for use by MR8C/4.

- Place a task in a finite time wait state by specifying a timeout value
  MR8C/4 guarantees that as stipulated in µITRON specification, timeout processing is not performed until a time equal to or greater than the specified timeout value elapses. More specifically, timeout processing is performed with the following timing.

  1. If the delay time value is 0 (for only dly_tsk)
     The task times out at the first time tick after the service call is issued.
  2. If the delay time value is a multiple of time tick interval
     The timer times out at the (timeout value / time tick interval) + first time tick. For example, if the time tick interval is 10 ms and the specified timeout value is 40 ms, then the timer times out at the fifth occurrence of the time tick. Similarly, if the time tick interval is 5 ms and the specified timeout value is 15 ms, then the timer times out at the fourth occurrence of the time tick.
  3. If the delay time value is not a multiple of time tick interval
     The timer times out at the (timeout value / time tick interval) + second time tick. For example, if the time tick interval is 10 ms and the specified timeout value is 35 ms, then the timer times out at the fifth occurrence of the time tick.

4.2.7 Cyclic Handler Function

The cyclic handler is a time event handler that is started every startup cycle after a specified startup phase has elapsed.

The cyclic handler may be started with or without saving the startup phase. In the former case, the cyclic handler is started relative to the point in time at which it was generated. In the latter case, the cyclic handler is started relative to the point in time at which it started operating. Figure 4.13 and Figure 4.14 show typical operations of the cyclic handler.

If the startup cycle is shorter than the time tick interval, the cyclic handler is started only once every time tick supplied (processing equivalent to isig_tim). For example, if the time tick interval is 10 ms and the startup cycle is 3 ms and the cyclic handler has started operating when a time tick is supplied, then the cyclic handler is started every time tick.

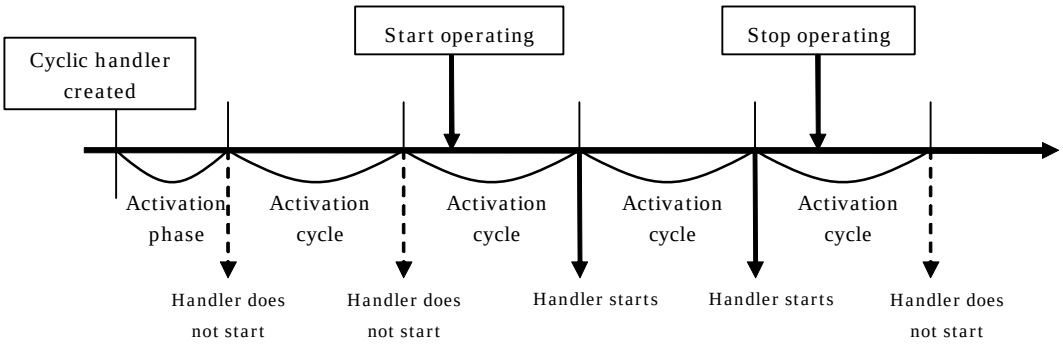


Figure 4.13 Cyclic handler operation in cases where the activation phase is saved

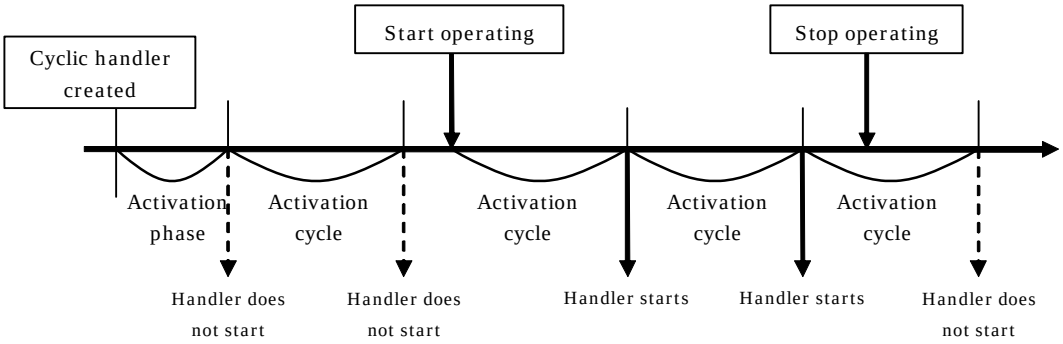


Figure 4.14 Cyclic handler operation in cases where the activation phase is not saved

- Start Cyclic Handler Operation (sta_cyc)
  Causes the cyclic handler with the specified ID to operational state.

- Stop Cyclic Handler Operation (stp_cyc)
  Causes the cyclic handler with the specified ID to non-operational state.

4.2.8 Alarm Handler Function

The alarm handler is a time event handler that is started only once at a specified time.

Use of the alarm handler makes it possible to perform time-dependent processing. The time of day is specified by a relative time. Figure 4.15 shows a typical operation of the alarm handler.

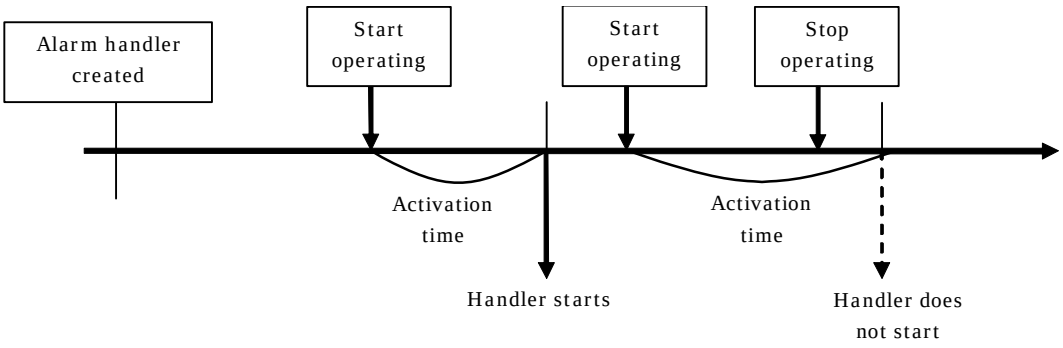


Figure 4.15 Typical operation of the alarm handler

- Start Alarm Handler Operation (sta_alm)
  Causes the alarm handler with the specified ID to operational state.

- Stop alarm Handler Operation (stp_alm)
  Causes the alarm handler with the specified ID to non-operational state.

4.2.9 System Status Management Function

- Reference task ID in the RUNNING state(get_tid)
  References the ID number of the task in the RUNNING state.
- Lock the CPU (loc_cpu)
  Places the system into a CPU locked state.
- Unlock the CPU (unl_cpu)
  Frees the system from a CPU locked state.
- Disable dispatching (dis_dsp)
  Places the system into a dispatching disabled state.
- Enable dispatching (ena_dsp)
  Frees the system from a dispatching disabled state.
- Reference context (sns_ctx)
  Gets the context status of the system.
- Reference CPU state (sns_loc)
  Gets the CPU lock status of the system.
- Reference dispatch disabling state (sns_dsp)
  Gets the dispatch disabling status of the system.

4.2.10 Interrupt Management Function

The interrupt management function provides a function to process requested external interrupts in real time.

The interrupt management service calls provided by the MR8C/4 kernel include the following:

- Returns from interrupt handler (ret_int)
  The ret_int service call activates the scheduler to switch over tasks as necessary when returning from the interrupt handler.
  When using the C language,[20], this function is automatically called at completion of the handler function. In this case, therefore, there is no need to invoke this service call.

Figure 4.16 shows an interrupt processing flow. Processing a series of operations from task selection to register restoration is called a "scheduler.".

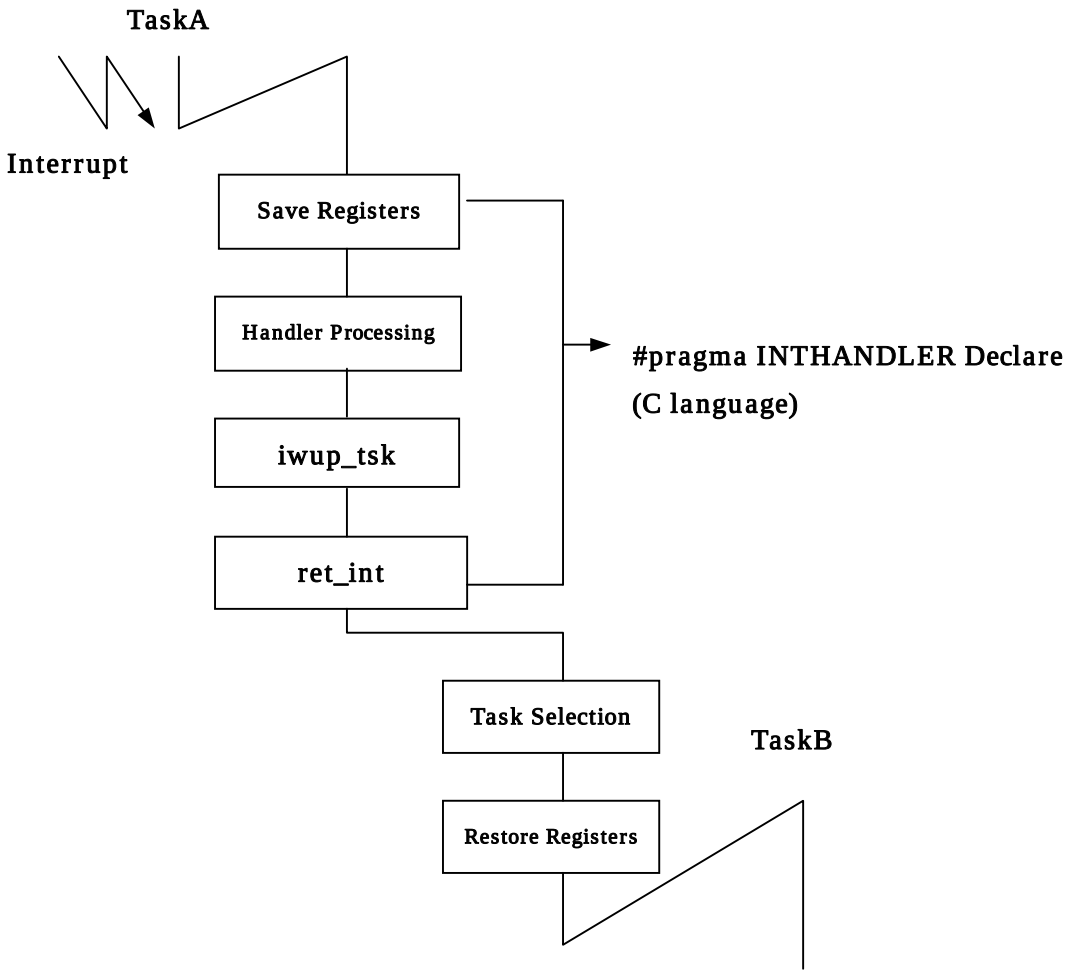


Figure 4.16 Interrupt process flow

[20] In the case that the interruput handler is specified by "#pragma INTHANDLER".

4.2.11 System Configuration Management Function

This function inspects the version information of MR8C/4.

- References Version Information(ref_ver)
  The ref_ver service call permits the user to get the version information of MR8C/4. This version information can be obtained in the standardized format of µITRON specification.

5. Service call reference

5.1 Task Management Function

Specifications of the task management function of MR8C/4 are listed in Table 5.1 below. The task description languages in item No. 4 are those specified in the GUI configurator. They are not output to a configuration file, nor are the MR8C/4 kernel concerned with them.

The task stack permits a section name to be specified for each task individually.

Table 5.1 Specifications of the Task Management Function

No.	Item	Content
1	Task ID	1-255
2	Task priority	1-255
3	Maximum number of activation request count	15
4	Task attribute	TA_HLNG : Tasks written in high-level language TA_ASM : Tasks written in as-sem-bly language TA_ACT : Startup attribute
5	Task stack	Section specifiable

Table 5.2 List of Task Management Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	sta_tsk	**[B]**	Starts task and specifies start code	O		O	O	O	
2	ista_tsk				O	O	O	O	
3	ext_tsk	**[S][B]**	Exits current task	O		O	O	O	O
4	ter_tsk	**[S][B]**	Forcibly terminates a task	O		O	O	O	
5	chg_pri	**[S][B]**	Changes task priority	O		O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls

- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

sta_tsk　　Activate task with a start code
ista_tsk　　Activate task with a start code (handler only)

[[C Language API]]

```
ER ercd = sta_tsk( ID tskid,VP_INT stacd );
ER ercd = ista_tsk ( ID tskid,VP_INT stacd );
```

- **Parameters**

ID	tskid	ID number of the target task
VP_INT	stacd	Task start code

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr8c.inc
sta_tsk  TSKID,STACD
ista_tsk TSKID,STACD
```

- **Parameters**

TSKID	ID number of the target task
STATCD	Task start code

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R1	Task start code
A0	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid (task indicated by tskid is not DOMANT state)

[[Functional description]]

This service call starts the task indicated by tskid. In other words, it places the specified task from DORMANT state into READY state or RUNNING state. This service call does not enqueue task activation requests. Therefore, if a task activation request is issued while the target task is not DORMANT state, the error code E_OBJ is returned to the service call issuing task. This service call is effective only when the specified task is in DORMANT state. The task start code stacd is 16 bits long. This task start code is passed as parameter to the activated task.

If a task is restarted that was once terminated by ter_tsk or ext_tsk, the task performs the following as it starts up.

1. Initializes the current priority of the task.

2. Clears the number of queued wakeup requests.

3. Clears the number of nested forcible wait requests.

If this service call is to be issued from task context, use sta_tsk; if issued from non-task context, use ista_tsk.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
    ER ercd;
    VP_INT stacd = 0;
    ercd = sta_tsk( ID_task2, stacd );
          :
  }
  void task2(VP_INT msg)
  {
    if(msg == 0)
          :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB        task
task:
                :
    PUSHM       A0,R1,R3
    sta_tsk     #ID_TASK4,#012345678H
                :
```

ext_tsk Terminate invoking task

[[C Language API]]

```
ER ercd = ext_tsk();
```

- **Parameters**

  None

- **Return Parameters**

  Not return from this service call

[[Assembly language API]]

```
.include mr8c.inc
ext_tsk
```

- **Parameters**

  None

- **Register contents after service call is issued**

  Not return from this service call

[[Error code]]

Not return from this service call

[[Functional description]]

This service call terminates the invoking task. In other words, it places the issuing task from RUNNING state into DORMANT state. However,

This service call is designed to be issued automatically at return from a task.

In the invocation of this service call, the resources the issuing task had acquired previously (e.g., semaphore) are not released.

This service call can only be used in task context. This service call can be used even in a CPU locked state, but cannot be used in non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task(void)
  {
       :
    ext_tsk();
  }
```

<<Example statement in assembly language>>

```
    .INCLUDE    mr8c.inc
    .GLB        task
task:
        :
    ext_tsk
```

ter_tsk Terminate task

[[C Language API]]

```
ER ercd = ter_tsk( ID tskid );
```

- **Parameters**

ID	tskid	ID number of the forcibly terminated task

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr8c.inc
ter_tsk  TSKID
```

- **Parameters**

TSKID	ID number of the forcibly terminated task

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A0	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is an inactive state)
E_ILUSE	Service call improperly used task indicated by tskid is the issuing task itself)

[[Functional description]]

This service call terminates the task indicated by tskid.

If a task specifies its own task ID or TSK_SELF, an E_ILUSE error is returned.

If the specified task was placed into WAITING state and has been enqueued in some waiting queue, the task is dequeued from it by execution of this service call. However, the semaphore and other resources the specified task had acquired previously are not released.

If the task indicated by tskid is in DORMANT state, it returns the error code E_OBJ as a return value for the service call.

This service call can only be used in task context, and cannot be used in non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
          :
      ter_tsk( ID_main );
          :
  }
```

<<Example statement in assembly language>>

```
     .INCLUDE    mr8c.inc
     .GLB        task
task:
                 :
     PUSHM       A0
     ter_tsk     #ID_TASK3
                 :
```

chg_pri Change task priority

[[C Language API]]

```
ER ercd = chg_pri( ID tskid, PRI tskpri );
```

- **Parameters**

ID	tskid	ID number of the target task
PRI	tskpri	Priority of the target task

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr8c.inc
chg_pri  TSKID,TSKPRI
```

- **Parameters**

TSKID	ID number of the target task
TSKPRI	Priority of the target task

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R3	Priority of the target task
A0	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is an inactive state)

[[Functional description]]

This service call changes the priority of the task indicated by tskid to the value indicated by tskpri, and performs rescheduling based on the result of that priority change. Therefore, if this service call is executed on a task enqueued in a ready queue (including one that is in an executing state) or a task in a waiting queue in which tasks are enqueued in order of priority, the target task is moved to behind the tail of a relevant priority part of the queue. Even when the same priority as the previous one is specified, the task is moved to behind the tail of the queue.

The smaller the number, the higher the task priority, with 1 assigned the highest priority. The minimum value specifiable as priority is 1. The specifiable maximum value of priority is the maximum value of priority specified in a configuration file, providing that it is within the range 1 to 255. For example, if system specification in a configuration file is as follows,

```
system{
        stack_size    = 0x100;
        priority      = 13;
};
```

then priority can be specified in the range 1 to 13.

If TSK_SELF is specified, the priority of the issuing task is changed. If TSK_SELF is specified for tskid in non-task context, operation of the service call cannot be guaranteed. If TPRI_INI is specified, the task has its priority changed to the initial priority that was specified when the task was created. The changed task priority remains effective until the task is terminated or this service call is executed again.

If the task indicated by tskid is in DORMANT state, it returns the error code E_OBJ as a return value for the service call. Since the MR8C/4 does not support the mutex function, in no case will the error code E_ILUSE be returned.

If this service call is to be issued from task context, use chg_pri.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
          :
      chg_pri( ID_task2, 2 );
          :
  }
```

<<Example statement in assembly language>>

```
    .Include mr8c.inc
    .GLB        task
task:
                :
    pushm       A0,R3
    chg_pri     #ID_TASK3,#1
                :
```

5.2 Task Dependent Synchronization Function

Specifications of the task-dependent synchronization function are listed in below.

Table 5.3 Specifications of the Task Dependent Synchronization Function

No.	Item	Content
1	Maximum value of task wakeup request count	15
2	Maximum number of nested forcible task wait requests count	1

Table 5.4 List of Task Dependent Synchronization Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	slp_tsk	**[S][B]**	Puts task to sleep	O		O		O	
2	wup_tsk	**[S][B]**	Wakes up task	O		O	O	O	
3	iwup_tsk	**[S][B]**			O	O	O	O	
4	can_wup		Cancels wakeup request	O		O	O	O	
5	rel_wai	**[S][B]**	Releases task from waiting	O		O	O	O	
6	irel_wai	**[S][B]**			O	O	O	O	
7	sus_tsk	**[S][B]**	Suspends task	O		O	O	O	
8	rsm_tsk	**[S][B]**	Resumes suspended task	O		O	O	O	
9	dly_tsk	**[S][B]**	Delays task	O		O		O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls

- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

slp_tsk　　Put task to sleep

[[C Language API]]

```
ER ercd = slp_tsk();
```

- **Parameters**

  None

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr8c.inc
slp_tsk
```

- **Parameters**

  None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code

[[Error code]]

E_RLWAI	Forced release from waiting

[[Functional description]]

This service call places the issuing task itself from RUNNING state into sleeping wait state. The task placed into WAITING state by execution of this service call is released from the wait state in the following cases:

- **When a task wakeup service call is issued from another task or an interrupt**
  The error code returned in this case is E_OK.
- **When a forcible awaking service call is issued from another task or an interrupt**
  The error code returned in this case is E_RLWAI.

If the task receives sus_tsk issued from another task while it has been placed into WAITING state by this service call, it goes to WAITING-SUSPENDED state. In this case, even when the task is released from WAITING state by a task wakeup service call, it still remains in SUSPENDED state, and its execution cannot be resumed until rsm_tsk is issued.

This service call can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
                :
       if( slp_tsk() != E_OK )
           error("Forced wakeup\n");
                :
                :
  }
```

<<Example statement in assembly language>>

```
     .include mr8c.inc
     .GLB        task
task:
           :
     slp_tsk
           :
```

wup_tsk Wakeup task
iwup_tsk Wakeup task (handler only)

[[C Language API]]

```
ER ercd = wup_tsk( ID tskid );
ER ercd = iwup_tsk( ID tskid );
```

- **Parameters**

ID	tskid	ID number of the target task

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr8c.inc
wup_tsk  TSKID
iwup_tsk  TSKID
```

- **Parameters**

TSKID	ID number of the target task

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A0	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is an inactive state)
E_QOVR	Queuing overflow

[[Functional description]]

If the task specified by tskid has been placed into WAITING state by slp_tsk this service call wakes up the task from WAITING state to place it into READY or RUNNING state. Or if the task specified by tskid is in WAITING-SUSPENDED state, this service call awakes the task from only the sleeping state so that the task goes to SUSPENDED state.

If a wakeup request is issued while the target task remains in DORMANT state, the error code E_OBJ is returned to the service call issuing task. If TSK_SELF is specified for tskid, it means specifying the issuing task itself. If TSK_SELF is specified for tskid in non-task context, operation of the service call cannot be guaranteed.

If this service call is issued to a task that has not been placed in WAITING state or in WAITING-SUSPENDED state by execution of slp_tsk, the wakeup request is accumulated. More specifically, the wakeup request count for the target task to be awakened is incremented by 1, in which way wakeup requests are accumulated.

The maximum value of the wakeup request count is 15. If while the wakeup request count = 15 a new wakeup request is generated exceeding this limit, the error code E_QOVR is returned to the task that issued the service call, with the wakeup request count left intact.

If this service call is to be issued from task context, use wup_tsk; if issued from non-task context, use iwup_tsk.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
                :
    if( wup_tsk( ID_main ) != E_OK )
        printf("Can't wakeup main()\n");
                :
  }
```

<<Example statement in assembly language>>

```
     .include mr8c.inc
     .GLB        task
task:
           :
     PUSHM       A0
     wup_tsk     #ID_TASK1
           :
```

can_wup Cancel wakeup request

[[C Language API]]

```
ER_UINT wupcnt = can_wup( ID tskid );
```

- **Parameters**

ID	tskid	ID number of the target task

- **Return Parameters**

ER_UINT	wupcnt > 0	Canceled wakeup request count
	wupcnt <0	Error code

[[Assembly language API]]

```
.include mr8c.inc
can_wup  TSKID
```

- **Parameters**

TSKID	ID number of the target task

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code,Canceled wakeup request count
A0	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is an inactive state)

[[Functional description]]

This service call clears the wakeup request count of the target task indicated by tskid to 0. This means that because the target task was in either WAITING state nor WAITING-SUSPENDED state when an attempt was made to wake it up by wup_tsk or iwup_tsk before this service call was issued, the attempt resulted in only accumulating wakeup requests and this service call clears all of those accumulated wakeup requests.

Furthermore, the wakeup request count before being cleared to 0 by this service call, i.e., the number of wakeup requests that were issued in vain (wupcnt) is returned to the issuing task. If a wakeup request is issued while the target task is in DORMANT state, the error code E_OBJ is returned. If TSK_SELF is specified for tskid, it means specifying the issuing task itself.

This service call can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
      ER_UINT wupcnt;
                :
      wupcnt = can_wup(ID_main);
      if( wup_cnt  > 0 )
          printf("wupcnt = %d\n",wupcnt);
                :
  }
```

<<Example statement in assembly language>>

```
     .include mr8c.inc
     .GLB        task
task:
           :
     PUSHM       A0
     can_wup     #ID_TASK3
           :
```

rel_wai Release task from waiting
irel_wai Release task from waiting (handler only)

[[C Language API]]

```
ER ercd = rel_wai( ID tskid );
ER ercd = irel_wai( ID tskid );
```

- **Parameters**

ID	tskid	ID number of the target task

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr8c.inc
rel_wai  TSKID
irel_wai  TSKID
```

- **Parameters**

TSKID	ID number of the target task

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A0	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is not an wait state)

[[Functional description]]

This service call forcibly release the task indicated by tskid from waiting (except SUSPENDED state) to place it into READY or RUNNING state. The forcibly released task returns the error code E_RLWAI. If the target task has been enqueued in some waiting queue, the task is dequeued from it by execution of this service call.

If this service call is issued to a task in WAITING-SUSPENDED state, the target task is released from WAITING state and goes to SUSPENDED state.[21]

If the target task is not in WAITING state, the error code E_OBJ is returned. This service call forbids specifying the issuing task itself for tskid.

If this service call is to be issued from task context, use rel_wai; if issued from non-task context, use irel_wai.

[21] This means that tasks cannot be resumed from SUSPENDED state by this service call. Only the rsm_tsk, irsm_tsk, frsm_tsk, and ifrsm_tsk service calls can release them from SUSPENDED state.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
              :
       if( rel_wai( ID_main ) != E_OK )
           error("Can't rel_wai main()\n");
            :
  }
```

<<Example statement in assembly language>>

```
     .include mr8c.inc
     .GLB        task
task:
            :
     PUSHM       A0
     rel_wai     #ID_TASK2
            :
```

sus_tsk Suspend task

[[C Language API]]

```
ER ercd = sus_tsk( ID tskid );
```

- **Parameters**

ID	tskid	ID number of the target task

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr8c.inc
sus_tsk  TSKID
```

- **Parameters**

TSKID	ID number of the target task

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A0	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is an inactive state)
E_QOVR	Queuing overflow

[[Functional description]]

This service call aborts execution of the task indicated by tskid and places it into SUSPENDED state. Tasks are resumed from this SUSPENDED state by the rsm_tsk service call. If the task indicated by tskid is in DORMANT state, it returns the error code E_OBJ as a return value for the service call.

The maximum number of suspension requests by this service call that can be nested is 1. If this service call is issued to a task which is already in SUSPENDED state, the error code E_QOVR is returned.

This service call forbids specifying the issuing task itself for tskid.

This service call can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
                :
       if( sus_tsk( ID_main ) != E_OK )
           printf("Can't suspend task main()\n");
                :
  }
```

<<Example statement in assembly language>>

```
     .include mr8c.inc
     .GLB        task
task:
           :
     PUSHM       A0
     sus_tsk     #ID_TASK2
           :
```

rsm_tsk Resume suspended task

[[C Language API]]

```
ER ercd = rsm_tsk( ID tskid );
```

- **Parameters**

ID	tskid	ID number of the target task

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr8c.inc
rsm_tsk  TSKID
```

- **Parameters**

TSKID	ID number of the target task

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A0	ID number of the target task

[[Error code]]

E_OBJ	Object status invalid(task indicated by tskid is not a forcible wait state)

[[Functional description]]

If the task indicated by tskid has been aborted by sus_tsk, this service call resumes the target task from SUSPENDED state. In this case, the target task is linked to behind the tail of the ready queue.

If a request is issued while the target task is not in SUSPENDED state (including DORMANT state), the error code E_OBJ is returned to the service call issuing task.

The rsm_tsk service call each operate the same way, because the maximum number of forcible wait requests that can be nested is 1.

This service call can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task1()
  {
                :
       if( rsm_tsk( ID_main ) != E_OK )
           printf("Can't resume main()\n");
                :
  }
```

<<Example statement in assembly language>>

```
     .include mr8c.inc
     .GLB        task
task:
           :
     PUSHM       A0
     rsm_tsk     #ID_TASK2
           :
```

dly_tsk Delay task

[[C Language API]]

```
ER ercd = dly_tsk(RELTIM dlytim);
```

- **Parameters**

RELTIM	dlytim	Delay time

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr8c.inc
dly_tsk  RELTIM
```

- **Parameters**

RELTIM	Delay time

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R1	Delay time (16 low-order bits)
R3	Delay time (16 high-order bits)

[[Error code]]

E_RLWAI	Forced release from waiting

[[Functional description]]

This service call temporarily stops execution of the issuing task itself for a duration of time specified by dlytim to place the task from RUNNING state into WAITING state. In this case, the task is released from the WAITING state at the first time tick after the time specified by dlytim has elapsed. Therefore, if specified dlytim = 0, the task is placed into WAITING state briefly and then released from the WAITING state at the first time tick.

The task placed into WAITING state by invocation of this service call is released from the WAITING state in the following cases. Note that when released from WAITING state, the task that issued the service call is removed from the timeout waiting queue and linked to a ready queue.

- **When the first time tick occurred after dlytim elapsed**
  The error code returned in this case is E_OK.
- **When the rel_wai or irel_wai service call is issued before dlytim elapses**
  The error code returned in this case is E_RLWAI.

Note that even when the wup_tsk or iwup_tsk service call is issued during the delay time, the task is not released from WAITNG state.

The delay time dlytim is expressed in ms units. Therefore, if specified as dly_tsk(50);, the issuing task is placed from RUNNING state into a delayed wait state for a period of 50 ms.

The values specified for dlytim must be within 0x7fffffff - time tick. If any value exceeding this limit is specified, the service call may not operate correctly.

This service call can be issued only from task context. It cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
                :
       if( dly_tsk() != E_OK )
           error("Forced wakeup\n");
                :
  }
```

<<Example statement in assembly language>>

```
     .include mr8c.inc
     .GLB        task
task:
           :
     PUSHM       R1,R3
     dly_tsk     #500
           :
```

5.3 Synchronization & Communication Function (Semaphore)

Specifications of the semaphore function of MR8C/4 are listed in Table 5.5.

Table 5.5 Specifications of the Semaphore Function

No.	Item	Content
1	Semaphore ID	1-255
2	Maximum number of resources	1-65535
3	Semaphore attribute	TA_FIFO: Tasks enqueued in order of FIFO

Table 5.6 List of Semaphore Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	sig_sem	**[S][B]**	Releases semaphore resource	O		O	O	O	
2	isig_sem	**[S][B]**			O	O	O	O	
3	wai_sem	**[S][B]**	Acquires semaphore resource	O		O		O	
4	pol_sem	**[S][B]**	Acquires semaphore resource(polling)	O		O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls

- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

sig_sem Release semaphore resource
isig_sem Release semaphore resource (handler only)

[[C Language API]]

```
ER ercd = sig_sem( ID semid );
ER ercd = isig_sem( ID semid );
```

- **Parameters**

ID	semid	Semaphore ID number to which returned

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr8c.inc
sig_sem SEMID
isig_sem SEMID
```

- **Parameters**

SEMID	Semaphore ID number to which returned

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A0	Semaphore ID number to which returned

[[Error code]]

E_QOVR	Queuing overflow

[[Functional description]]

This service call releases one resource to the semaphore indicated by semid.

If tasks are enqueued in a waiting queue for the target semaphore, the task at the top of the queue is placed into READY state. Conversely, if no tasks are enqueued in that waiting queue, the semaphore resource count is incremented by 1. If an attempt is made to return resources (sig_sem or isig_sem service call) causing the semaphore resource count value to exceed the maximum value specified in a configuration file (maxsem), the error code E_QOVR is returned to the service call issuing task, with the semaphore count value left intact.

If this service call is to be issued from task context, use sig_sem; if issued from non-task context, use isig_sem.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
                :
       if( sig_sem( ID_sem ) == E_QOVR )
         error("Overflow\n");
                :
  }
```

<<Example statement in assembly language>>

```
     .include mr8c.inc
     .GLB        task
task:
           :
     PUSHM       A0
     sig_sem     #ID_SEM2
           :
```

wai_sem Acquire semaphore resource
pol_sem Acquire semaphore resource (polling)

[[C Language API]]

```
ER ercd = wai_sem( ID semid );
ER ercd = pol_sem( ID semid );
```

- **Parameters**

ID	semid	Semaphore ID number to be acquired

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr8c.inc
wai_sem SEMID
pol_sem SEMID
```

- **Parameters**

SEMID	Semaphore ID number to be acquired

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A0	Semaphore ID number to be acquired

[[Error code]]

E_RLWAI	Forced release from waiting
E_TMOUT	Polling failure

[[Functional description]]

This service call acquires one semaphore resource from the semaphore indicated by semid.

If the semaphore resource count is equal to or greater than 1, the semaphore resource count is decremented by 1, and the service call issuing task continues execution. On the other hand, if the semaphore count value is 0, the wai_sem service call invoking task is enqueued in a waiting queue for that semaphore in order of FIFO. For the pol_sem service call, the task returns immediately and responds to the call with the error code E_TMOUT.

The task placed into WAITING state by execution of the wai_sem service call is released from the WAITING state in the following cases:

- **When the sig_sem or isig_sem service call is issued with task-awaking conditions thereby satisfied**
  The error code returned in this case is E_OK.
- **When the task is forcibly released from WAITING state by the rel_wai or irel_wai service call issued from another task or a handler**
  The error code returned in this case is E_RLWAI.

These service calls can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
                :
     if( wai_sem( ID_sem ) != E_OK )
         printf("Forced wakeup\n");
                :
     if( pol_sem( ID_sem ) != E_OK )
         printf("Timeout\n");
                :
  }
```

<<Example statement in assembly language>>

```
     .include mr8c.inc
     .GLB        task
task:
           :
     PUSHM       A0
     pol_sem     #ID_SEM1
           :
     PUSHM       A0
     wai_sem     #ID_SEM2
           :
```

5.4 Synchronization & Communication Function (Eventflag)

Specifications of the eventflag function of MR8C/4 are listed in Table 5.7.

Table 5.7 Specifications of the Eventflag Function

No.	Item	Content	
1	Event0flag ID	1-255	
2	Number of bits comprising eventflag	16 bits	
3	Eventflag attribute	TA_TFIFO:	Waiting tasks enqueued in order of FIFO
		TA_WSGL:	Multiple tasks cannot be kept waiting
		TA_WMUL:	Multiple tasks can be kept waiting
		TA_CLR:	Bit pattern cleared when waiting task is released

Table 5.8 List of Eventflag Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	set_flg	**[S][B]**	Sets eventflag	O		O	O	O	
2	iset_flg	**[S][B]**			O	O	O	O	
3	clr_flg	**[S][B]**	Clears eventflag	O		O	O	O	
4	wai_flg	**[S][B]**	Waits for eventflag	O		O		O	
5	pol_flg	**[S][B]**	Waits for eventflag (polling)	O		O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls

- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

set_flg Set eventflag
iset_flg Set eventflag (handler only)

[[C Language API]]

```
ER ercd = set_flg( ID flgid, FLGPTN setptn );
ER ercd = iset_flg( ID flgid, FLGPTN setptn );
```

- **Parameters**

ID	flgid	ID number of the eventflag to be set
FLGPTN	setptn	Bit pattern to be set

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr8c.inc
set_flg FLGID,SETPTN
iset_flg FLGID,SETPTN
```

- **Parameters**

FLGID	ID number of the eventflag to be set
SETPTN	Bit pattern to be set

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R3	Bit pattern to be set
A0	Eventflag ID number

[[Error code]]

None

[[Functional description]]

Of the 16-bit eventflag indicated by flgid, this service call sets the bits indicated by setptn. In other words, the value of the eventflag indicated by flgid is OR'd with setptn. If the alteration of the eventflag value results in task-awaking conditions for a task that has been kept waiting for the eventflag by the wai_flg or twai_flg service call becoming satisfied, the task is released from WAITING state and placed into READY or RUNNING state.

Task-awaking conditions are evaluated sequentially beginning with the top of the waiting queue. If TA_WMUL is specified as an eventflag attribute, multiple tasks kept waiting for the eventflag can be released from WAITING state at the same time by one set_flg or iset_flg service call issued. Furthermore, if TA_CLR is specified for the attribute of the target event-flag, all bit patterns of the eventflag are cleared, with which processing of the service call is terminated.

If all bits specified in setptn are 0, no operation will be performed for the target eventflag, in which case no errors are assumed, however.

If this service call is to be issued from task context, use set_flg; if issued from non-task context, use iset_flg.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task(void)
  {
                :
    set_flg( ID_flg,(FLGPTN)0xff00 );
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB        task
task:
          :
    PUSHM       A0, R3
    set_flg     #ID_FLG3,#0ff00H
          :
```

clr_flg Clear eventflag

[[C Language API]]

```
ER ercd = clr_flg( ID flgid, FLGPTN clrptn );
```

- **Parameters**

ID	flgid	ID number of the eventflag to be cleared
FLGPTN	clrptn	Bit pattern to be cleared

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr8c.inc
clr_flg FLGID,CLRPTN
```

- **Parameters**

FLGID	ID number of the eventflag to be cleared
CLRPTN	Bit pattern to be cleared

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A0	ID number of the eventflag to be cleared
R3	Bit pattern to be cleared

[[Error code]]

None

[[Functional description]]

Of the 16-bit eventflag indicated by flgid, this service call clears the bits whose corresponding values in clrptn are 0. In other words, the eventflag bit pattern indicated by flgid is updated by AND'ing it with clrptn. If all bits specified in clrptn are 1, no operation will be performed for the target eventflag, in which case no errors are assumed, however.

This service call can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task(void)
  {
                :
    clr_flg( ID_flg,(FLGPTN) 0xf0f0);
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB        task
task:
          :
    PUSHM       A0, R3
    clr_flg     #ID_FLG1,#0f0f0H
          :
```

wai_flg Wait for eventflag
pol_flg Wait for eventflag(polling)

[[C Language API]]

```
ER ercd = wai_flg( ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn );
ER ercd = pol_flg( ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn );
```

- **Parameters**

ID	flgid	ID number of the eventflag waited for
FLGPTN	waiptn	Wait bit pattern
MODE	wfmode	Wait mode
FLGPTN	*p_flgptn	Pointer to the area to which bit pattern is returned when released from wait

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code
FLGPTN	*p_flgptn	Pointer to the area to which bit pattern is returned when released from wait

[[Assembly language API]]

```
.include mr8c.inc
wai_flg   FLGID, WAIPTN, WFMODE
pol_flg   FLGID, WAIPTN, WFMODE
```

- **Parameters**

FLGID	ID number of the eventflag waited for
WAIPTN	Wait bit pattern
WFMODE	Wait mode

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R1	Wait mode
R2	bit pattern is returned when released from wait
R3	Wait bit pattern
A0	ID number of the eventflag waited for

[[Error code]]

E_RLWAI	Forced release from waiting
E_TMOUT	Polling failure
E_ILUSE	Service call improperly used (Tasks present waiting for TA_WSGL attribute eventflag)

[[Functional description]]

This service call waits until the eventflag indicated by flgid has its bits specified by waiptn set according to task-awaking conditions indicated by wfmode. Returned to the area pointed to by p_flgptn is the eventflag bit pattern at the time the task is released from WAITING state.

If the target eventflag has the TA_WSGL attribute and there are already other tasks waiting for the eventflag, the error code E_ILUSE is returned.

If task-awaking conditions have already been met when this service call is invoked, the task returns immediately and responds to the call with E_OK. If task-awaking conditions are not met and the invoked service call is wai_flg, the task is enqueued in an eventflag waiting queue in order of FIFO. For the pol_flg service call, the task returns immediately and responds to the call with the error code E_TMOUT.

The task placed into a wait state by execution of the wai_flg service call is released from WAITING state in the following cases:

- **When task-awaking conditions are met before the tmout time elapses**
  The error code returned in this case is E_OK.
- **When the task is forcibly released from WAITING state by the rel_wai or irel_wai service call issued from another task or a handler**
  The error code returned in this case is E_RLWAI.

The following shows how wfmode is specified and the meaning of each mode.

wfmdoe (wait mode)	Meaning
TWF_ANDW	Wait until all bits specified by waiptn are set (wait for the bits AND'ed)
TWF_ORW	Wait until one of the bits specified by waiptn is set (wait for the bits OR'ed)

These service calls can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
    UINT flgptn;
              :
      if(wai_flg(ID_flg2, (FLGPTN)0x0ff0, TWF_ANDW, &flgptn)!=E_OK)
        error("Wait Released\n");
              :
              :
      if(pol_flg(ID_flg2, (FLGPTN)0x0ff0, TWF_ORW, &flgptn)!=E_OK)
        printf("Not set EventFlag\n");
              :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB        task
task:
         :
    PUSHM       A0,R1,R3
    wai_flg     #ID_FLG1,#0003H,#TWF_ANDW
         :
    PUSHM       A0,R1,R3
    pol_flg     #ID_FLG2,#0008H,#TWF_ORW
         :
```

5.5 Synchronization & Communication Function (Data Queue)

Specifications of the data queue function of MR8C/4 are listed in Table 5.9.

Table 5.9 Specifications of the Data Queue Function

No.	Item	Content
1	Data queue ID	1-255
2	Capacity (data bytes) in data queue area	0-65535
3	Data size	16 bits
4	Data queue attribute	TA_TFIFO: Waiting tasks enqueued in order of FIFO

Table 5.10 List of Dataqueue Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	snd_dtq	**[S]**	Sends to data queue	O		O		O	
2	psnd_dtq	**[S]**	Sends to data queue (polling)		O	O	O	O	
3	ipsnd_dtq	**[S]**		O		O	O	O	
4	rcv_dtq	**[S]**	Receives from data queue		O	O		O	
5	prcv_dtq	**[S]**	Receives from data queue (polling)	O		O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls

- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

snd_dtq Send to data queue
psnd_dtq Send to data queue (polling)
ipsnd_dtq Send to data queue (polling, handler only)

[[C Language API]]

```
ER ercd = snd_dtq( ID dtqid, VP_INT data );
ER ercd = psnd_dtq( ID dtqid, VP_INT data );
ER ercd = ipsnd_dtq( ID dtqid, VP_INT data );
```

● Parameters

ID	dtqid	ID number of the data queue to which transmitted
VP_INT	data	Data to be transmitted

● Return Parameters

ER	ercd	Terminated normally (E_OK) or error code

[[Assembly language API]]

```
.include mr8c.inc
snd_dtq  DTQID, DTQDATA
psnd_dtq DTQID, DTQDATA
ipsnd_dtq DTQID, DTQDATA
```

● Parameters

DTQID	ID number of the data queue to which transmitted
DTQDATA	Data to be transmitted

● Register contents after service call is issued

Register name	Content after service call is issued
R0	Error code
R1	Data to be transmitted
A0	ID number of the data queue to which transmitted

[[Error code]]

E_RLWAI	Forced release from waiting
E_TMOUT	Polling failure

[[Functional description]]

This service call sends the 2-byte data indicated by data to the data queue indicated by dtqid. If any task is kept waiting for reception in the target data queue, the data is not stored in the data queue and instead sent to the task at the top of the reception waiting queue, with which the task is released from the reception wait state.

On the other hand, if snd_dtq is issued for a data queue that is full of data, the task that issued the service call goes from RUNNING state to a data transmission wait state, and is enqueued in transmission waiting queue, kept waiting for the data queue to become available. In that case, if the attribute of the specified data queue is TA_TFIFO, the task is enqueued in order of FIFO. For psnd_dtq and ipsnd_dtq, the task returns immediately and responds to the call with the error code E_TMOUT.

If there are no tasks waiting for reception, nor is the data queue area filled, the transmitted data is stored in the data queue.

The task placed into WAITING state by execution of the snd_dtq service call is released from WAITING state in the following cases:

- **When the rcv_dtq or prcv_dtq service call is issued with task-awaking conditions thereby satisfied**
  The error code returned in this case is E_OK.
- **When the task is forcibly released from WAITING state by the rel_wai or irel_wai service call issued from another task or a handler**
  The error code returned in this case is E_RLWAI.

These service calls can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  VP_INT data[10];
  void task(void)
  {
                :
      if( snd_dtq( ID_dtq, data[0]) == E_RLWAI ){
        error("Forced released\n");
    }
                :
      if( psnd_dtq( ID_dtq, data[1])== E_TMOUT ){
        error("Timeout\n");
    }
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB        task
_g_dtq: .LWORD  12345678H
task:
          :
    PUSHM       R1,R3,A0
    psnd_dtq    #ID_DTQ2,#0FFFFH
          :
    PUSHM       R1,R3,A0
    snd_dtq     #ID_DTQ3,#0ABCDH
          :
```

rcv_dtq Receive from data queue
prcv_dtq Receive from data queue (polling)

[[C Language API]]

```
ER ercd = rcv_dtq( ID dtqid, VP_INT *p_data );
ER ercd = prcv_dtq( ID dtqid, VP_INT *p_data );
```

- **Parameters**

ID	dtqid	ID number of the data queue from which to receive
VP_INT	*p_data	Pointer to the start of the area in which received data is stored

- **Return Parameters**

ER	ercd	Terminated normally (E_OK) or error code
VP_INT	*p_data	Pointer to the start of the area in which received data is stored

[[Assembly language API]]

```
.include mr8c.inc
rcv_dtq  DTQID
prcv_dtq DTQID
```

- **Parameters**

DTQID	ID number of the data queue from which to receive

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R1	Received data
A0	Data queue ID number

[[Error code]]

E_RLWAI	Forced release from waiting
E_TMOUT	Polling failure

[[Functional description]]

This service call receives data from the data queue indicated by dtqid and stores the received data in the area pointed to by p_data. If data is present in the target data queue, the data at the top of the queue or the oldest data is received. This results in creating a free space in the data queue area, so that a task enqueued in a transmission waiting queue is released from WAITING state, and starts sending data to the data queue area.

If no data exist in the data queue and there is any task waiting to send data (i.e., data bytes in the data queue area = 0), data for the task at the top of the data transmission waiting queue is received. As a result, the task kept waiting to send that data is released from WAITING state.

On the other hand, if rcv_dtq is issued for the data queue which has no data stored in it, the task that issued the service call goes from RUNNING state to a data reception wait state, and is enqueued in a data reception waiting queue. At this time, the task is enqueued in order of FIFO. For the prcv_dtq service calls, the task returns immediately and responds to the call with the error code E_TMOUT.

The task placed into a wait state by execution of the rcv_dtq service call is released from the wait state in the following cases:

- **When the rcv_dtq or prcv_dtq service call is issued with task-awaking conditions thereby satisfied**
  The error code returned in this case is E_OK.
- **When the task is forcibly released from WAITING state by the rel_wai or irel_wai service call issued from another task or a handler**
  The error code returned in this case is E_RLWAI.

These service calls can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
                                          void task()
  {
    VP_INT data;
                :
      if( rcv_dtq( ID_dtq, &data ) != E_RLWAI )
          error("forced wakeup\n");
                :
      if( prcv_dtq( ID_dtq, &data ) != E_TMOUT )
          error("Timeout\n");
                :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB        task
task:
          :
    PUSHM       A0
    prcv_dtq    #ID_DTQ2
          :
    PUSHM       A0
    rcv_dtq     #ID_DTQ2
          :
```

5.6 Time Management Function

Table 5.11 List of Time Management Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	isig_tim	**[S]**	Supply a time tick		○	○	○	○	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls

- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

isig_tim Supply a time tick

[[Functional description]]

This service call updates the system time.

The isig_tim is automatically started every tick_time interval(ms) if the system clock is defined by the configuration file. The application cannot call this function because it is not implementing as service call.

When a time tick is supplied, the kernel is processed as follows:

(1) Updates the system time
(2) Starts an alarm handler
(3) Starts a cyclic handler
(4) Processes the timeout processing of the task put on WAITING state by dly_tsk service call with time-out.

5.7 Time Management Function (Cyclic Handler)

Specifications of the cyclic handler function of MR8C/4 are listed in Table 5.12. The cyclic handler description languages in item No. 4 are those specified in the GUI configurator. They are not output to a configuration file, nor are the MR8C/4 kernel concerned with them.

Table 5.12 Specifications of the Cyclic Handler Function

No.	Item	Content	
1	Cyclic handler ID	1-255	
2	Activation cycle	0-7fffffff[ms]	
3	Activation phase	0-7fffffff[ms]	
4	Extended information	16 bits	
5	Cyclic handler attribute	TA_HLNG:	Handlers written in high-level language
		TA_ASM:	Handlers written in assembly language
		TA_STA:	Starts operation of cyclic handler
		TA_PHS:	Saves activation phase

Table 5.13 List of Cyclic Handler Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	sta_cyc	**[S][B]**	Starts cyclic handler operation	O		O	O	O	
2	stp_cyc	**[S][B]**	Stops cyclic handler operation	O		O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls

- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

sta_cyc Start cyclic handler operation

[[C Language API]]

```
ER ercd = sta_cyc( ID cycid );
```

- **Parameters**

ID	cycid	ID number of the cyclic handler to be operated

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr8c.inc
sta_cyc  CYCNO
```

- **Parameters**

CYCNO	ID number of the cyclic handler to be operated

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A0	ID number of the cyclic handler to be operated

[[Error code]]

None

[[Functional description]]

This service call places the cyclic handler indicated by cycid into an operational state. If the cyclic handler attribute of TA_PHS is not specified, the cyclic handler is started every time the activate cycle elapses, start with the time at which this service call was invoked.

If while TA_PHS is not specified this service call is issued to a cyclic handler already in an operational state, it sets the time at which the cyclic handler is to start next.

If while TA_PHS is specified this service call is issued to a cyclic handler already in an operational state, it does not set the startup time.

This service call can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
        :
    sta_cyc ( ID_cyc1 );
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB       task
task:
          :
        PUSHM   A0
        sta_cyc #ID_CYC1
          :
```

stp_cyc Stops cyclic handler operation

[[C Language API]]

```
ER ercd = stp_cyc( ID cycid );
```

- **Parameters**

ID	cycid	ID number of the cyclic handler to be stopped

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr8c.inc
stp_cyc  CYCNO
```

- **Parameters**

CYCNO	ID number of the cyclic handler to be stopped

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A0	ID number of the cyclic handler to be stopped

[[Error code]]

None

[[Functional description]]

This service call places the cyclic handler indicated by cycid into a non-operational state.

This service call can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
        :
    stp_cyc ( ID_cyc1 );
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB       task
task:
         :
        PUSHM   A0
        stp_cyc #ID_CYC1
          :
```

5.8 Time Management Function (Alarm Handler)

Specifications of the alarm handler function of MR8C/4 are listed in Table 5.14. The alarm handler description languages in item No. 4 are those specified in the GUI configurator. They are not output to a configuration file, nor are the MR8C/4 kernel concerned with them.

Table 5.14 Specifications of the Alarm Handler Function

No.	Item	Content	
1	Alarm handler ID	1-255	
2	Activation time	0-7fffffff [ms]	
3	Extended information	16 bits	
4	Alarm handler attribute	TA_HLNG:	Handlers written in high-level language
		TA_ASM:	Handlers written in assembly language

Table 5.15 List of Alarm Handler Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	sta_alm		Starts alarm handler operation	○		○	○	○	
2	stp_alm		Stops alarm handler operation	○		○	○	○	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls

- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

sta_alm Start alarm handler operation

[[C Language API]]

```
ER ercd = sta_alm( ID almid, RELTIM almtim );
```

- **Parameters**

ID	almid	ID number of the alarm handler to be operated
RELTIM	almtim	Alarm handler startup time (relative time)

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr8c.inc
sta_alm  ALMID,ALMTIM
```

- **Parameters**

ALMID	ID number of the alarm handler to be operated
ALMTIM	Alarm handler startup time (relative time)

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
R1	Alarm handler startup time (relative time)
A0	ID number of the alarm handler to be operated

[[Error code]]

None

[[Functional description]]

This service call sets the activation time of the alarm handler indicated by almid as a relative time of day after the lapse of the time specified by almtim from the time at which it is invoked, and places the alarm handler into an operational state.

If an already operating alarm handler is specified, the previously set activation time is cleared and updated to a new activation time. If almtim = 0 is specified, the alarm handler starts at the next time tick. The values specified for almtim must be within (0x7fffffff – time tick). If any value exceeding this limit is specified, the service call may not operate correctly. If 0 is specified for almtim , the alarm handler is started at the next time tick.

This service call can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
        :
    sta_alm ( ID_alm1,100 );
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB        task
task:
          :
        PUSHM   A0
        sta_alm #ID_ALM1,#100
          :
```

stp_alm Stop alarm handler operation

[[C Language API]]

```
ER ercd = stp_alm( ID almid );
```

- **Parameters**

ID	almid	ID number of the alarm handler to be stopped

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr8c.inc
stp_alm  ALMID
```

- **Parameters**

ALMID	ID number of the alarm handler to be stopped

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A0	ID number of the alarm handler to be stopped

[[Error code]]

None

[[Functional description]]

This service call places the alarm handler indicated by almid into a non-operational state.

This service call can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
        :
    stp_alm ( ID_alm1 );
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB       task
task:
         :
        PUSHM   A0
        stp_alm #ID_ALM1
         :
```

5.9 System Status Management Function

Table 5.16 List of System Status Management Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
3	get_tid	**[S][B]**	References task ID in the RUNNING state	O		O	O	O	
5	loc_cpu	**[S][B]**	Locks the CPU	O		O	O	O	O
7	unl_cpu	**[S][B]**	Unlocks the CPU	O		O	O	O	O
9	dis_dsp	**[S][B]**	Disables dispatching	O		O	O	O	
10	ena_dsp	**[S][B]**	Enables dispatching	O		O	O	O	
11	sns_ctx	**[S]**	References context	O	O	O	O	O	O
12	sns_loc	**[S]**	References CPU state	O	O	O	O	O	O
13	sns_dsp	**[S]**	References dispatching state	O	O	O	O	O	O

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls

- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

get_tid Reference task ID in the RUNNING state

[[C Language API]]

```
ER ercd = get_tid( ID *p_tskid );
```

- **Parameters**

ID	*p_tskid	Pointer to task ID

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)
ID	*p_tskid	Pointer to task ID

[[Assembly language API]]

```
.include mr8c.inc
get_tid
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A0	Acquired task ID

[[Error code]]

None

[[Functional description]]

This service call returns the task ID currently in RUNNING state to the area pointed to by p_tskid. If this service call is issued from a task, the ID number of the issuing task is returned.

This service call can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
     ID tskid;
        :
      get_tid(&tskid);
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB       task
task:
          :
        PUSHM   A0
        get_tid
           :
```

loc_cpu Lock the CPU

[[C Language API]]

```
ER ercd = loc_cpu();
```

- **Parameters**

None

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr8c.inc
loc_cpu
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code

[[Error code]]

None

[[Functional description]]

This service call places the system into a CPU locked state, thereby disabling interrupts and task dispatches. The features of a CPU locked state are outlined below.

(1) No task scheduling is performed during a CPU locked state.
(2) No external interrupts are accepted unless their priority levels are higher than the kernel interrupt mask level defined in the configurator.
(3) Only the following service calls can be invoked from a CPU locked state. If any other service calls are invoked, operation of the service call cannot be guaranteed.
  * ext_tsk
  * loc_cpu
  * unl_cpu
  * sns_ctx
  * sns_loc
  * sns_dsp

The system is freed from a CPU locked state by one of the following operations.

(a) Invocation of the unl_cpu service call
(b) Invocation of the ext_tsk service call

Transitions between CPU locked and CPU unlocked states occur only when the loc_cpu, unl_cpu, or ext_tsk service call is invoked. The system must always be in a CPU unlocked state when the interrupt handler or the time event handler is terminated. If either handler terminates while the system is in a CPU locked state, handler operation cannot be guaranteed. Note that the system is always in a CPU unlocked state when these handlers start.

Invoking this service call again while the system is already in a CPU locked state does not cause an error, in which case task queuing is not performed, however.

This service call can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
        :
       loc_cpu();
        :
  }
```

<<Example statement in assembly language>>

```
     .include mr8c.inc
     .GLB        task
task:
          :
         loc_cpu
          :
```

unl_cpu　　Unlock the CPU

[[C Language API]]

```
ER ercd = unl_cpu();
```

- **Parameters**

None

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr8c.inc
unl_cpu
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code

[[Error code]]

None

[[Functional description]]

This service call frees the system from a CPU locked state that was set by the loc_cpu service call. If the unl_cpu service call is issued from a dispatching enabled state, task scheduling is performed.

The CPU locked state and the dispatching disabled state are managed independently of each other. Therefore, the system cannot be freed from a dispatching disabled state by the unl_cpu service call unless the ena_dsp service call is used.

This service call can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
        :
      unl_cpu();
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB        task
task:
          :
        unl_cpu
          :
```

dis_dsp Disable dispatching

[[C Language API]]

```
ER ercd = dis_dsp();
```

- **Parameters**

None

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr8c.inc
dis_dsp
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code

[[Error code]]

None

[[Functional description]]

This service call places the system into a dispatching disabled state. The features of a dispatching disabled state are outlined below.

(1) Since task scheduling is not performed anymore, no tasks other than the issuing task itself will be placed into RUNNING state.
(2) Interrupts are accepted.
(3) No service calls can be invoked that will place tasks into WAITING state.

If one of the following operations is performed during a dispatching disabled state, the system status returns to a task execution state.

(a) Invocation of the ena_dsp service call
(b) Invocation of the ext_tsk service call

Transitions between dispatching disabled and dispatching enabled states occur only when the dis_dsp, ena_dsp, or ext_tsk service call is invoked.

Invoking this service call again while the system is already in a dispatching disabled state does not cause an error, in which case task queuing is not performed, however.

This service call can be issued only from task context. It cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
        :
      dis_dsp();
        :
  }
```

<<Example statement in assembly language>>

```
     .include mr8c.inc
     .GLB        task
task:
          :
        dis_dsp
          :
```

ena_dsp Enables dispatching

[[C Language API]]

```
ER ercd = ena_dsp();
```

- **Parameters**

None

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr8c.inc
ena_dsp
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code

[[Error code]]

None

[[Functional description]]

This service call frees the system from a dispatching disabled state that was set by the dis_dsp service call. As a result, task scheduling is resumed when the system has entered a task execution state.

Invoking this service call from a task execution state does not cause an error, in which case task queuing is not performed, however.

This service call can be issued only from task context. It cannot be issued from non-task context.

[[Example program statement]]

```
<<Example statement in C language>>
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
        :
      ena_dsp();
        :
  }
<<Example statement in assembly language>>
     .include mr8c.inc
     .GLB        task
task:
           :
         ena_dsp
           :
```

sns_ctx Reference context

[[C Language API]]

```
BOOL state = sns_ctx();
```

- **Parameters**

None

- **Return Parameters**

BOOL	state	TRUE: Non-task context FALSE: Task context

[[Assembly language API]]

```
.include mr8c.inc
sns_ctx
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	TRUE:Non-Task context FALSE: Task context

[[Error code]]

None

[[Functional description]]

This service call returns TRUE when it is invoked from non-task context, or returns FALSE when invoked from task context. This service call can also be invoked from a CPU locked state.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
     BOOL stat;
        :
      stat = sns_ctx();
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB        task
task:
          :
        sns_ctx
          :
```

sns_loc Reference CPU state

[[C Language API]]

```
BOOL state = sns_loc();
```

- **Parameters**

None

- **Return Parameters**

BOOL	state	TRUE: CPU locked state FALSE: CPU unlocked state

[[Assembly language API]]

```
.include mr8c.inc
sns_loc
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	TRUE: CPU locked state FALSE:CPUCPU unlocked state

[[Error code]]

None

[[Functional description]]

This service call returns TRUE when the system is in a CPU locked state, or returns FALSE when the system is in a CPU unlocked state. This service call can also be invoked from a CPU locked state.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
     BOOL stat;
        :
      stat = sns_loc();
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB        task
task:
          :
        sns_loc
          :
```

sns_dsp Reference dispatching state

[[C Language API]]

```
BOOL state = sns_dsp();
```

- **Parameters**

None

- **Return Parameters**

BOOL	state	TRUE: Dispatching disabled state FALSE: Dispatching enabled state

[[Assembly language API]]

```
.include mr8c.inc
sns_dsp
```

- **Parameters**

None

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	TRUE: Dispatching disabled state FALSE: Dispatching enabled state

[[Error code]]

None

[[Functional description]]

This service call returns TRUE when the system is in a dispatching disabled state, or returns FALSE when the system is in a dispatching enabled state. This service call can also be invoked from a CPU locked state.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
     BOOL stat;
        :
      stat = sns_dsp();
        :
  }
```

<<Example statement in assembly language>>

```
    .include mr8c.inc
    .GLB        task
task:
          :
        sns_dsp
          :
```

5.10 Interrupt Management Function

Table 5.17 List of Interrupt Management Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	ret_int		Returns from an interrupt handler		O	O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls

- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

ret_int — Returns from an interrupt handler (when written in assembly language)

[[C Language API]]

This service call cannot be written in C language.[22]

[[Assembly language API]]

```
.include mr8c.inc
ret_int
```

- **Parameters**

None

[[Error code]]

Not return to the interrupt handler that issued this service call.

[[Functional description]]

This service call performs the processing necessary to return from an interrupt handler. Depending on return processing, it activates the scheduler to switch tasks from one to another.

If this service call is executed in an interrupt handler, task switching does not occur, and task switching is postponed until the interrupt handler terminates.

However, if the ret_int service call is issued from an interrupt handler that was invoked from an interrupt that occurred within another interrupt, the scheduler is not activated. The scheduler is activated for interrupts from a task only.

When writing this service call in assembly language, be aware that the service call cannot be issued from a subroutine that is invoked from an interrupt handler entry routine. Always make sure this service call is executed in the entry routine or entry function of an interrupt handler. For example, a program like the one shown below may not operate normally.

```
.include mr8c.inc
/* NG */
.GLB intr
intr:
          jsr.b func
:
func:
          ret_int
```

Therefore, write the program as shown below.

```
.include mr8c.inc
/* OK */
.GLB intr
intr:
          jsr.b func
          ret_int
func:
:
          rts
```

Make sure this service call is issued from only an interrupt handler. If issued from a cyclic handler, alarm handler, or a task, this service call may not operate normally.

[22] If the starting function of an interrupt handler is declared by #pragma INTHANDLER, the ret_int service call is automatically issued at the exit of the function.

5.11 System Configuration Management Function

Table 5.18 List of System Configuration Management Function Service Call

No.	Service Call		Function	System State					
				T	N	E	D	U	L
1	ref_ver	**[S]**	References version information	O		O	O	O	

Notes:

- [S]: Standard profile service calls
  [B]: Basic profile service calls

- Each sign within " System State " is a following meaning.
  - T: Can be called from task context
  - N: Can be called from non-task context
  - E: Can be called from dispatch-enabled state
  - D: Can be called from dispatch-disabled state
  - U: Can be called from CPU-unlocked state
  - L: Can be called from CPU-locked state

ref_ver Reference version information

[[C Language API]]

```
ER ercd = ref_ver( T_RVER *pk_rver );
```

- **Parameters**

T_RVER	*pk_rver	Pointer to the packet to which version information is returned

Contents of pk_rver

typedef struct t_rver {

UH	maker	0	2	Kernel manufacturer code
UH	prid	+2	2	Kernel identification number
UH	spver	+4	2	ITRON specification version number
UH	prver	+6	2	Kernel version number
UH	prno[4]	+8	2	Kernel product management information

} T_RVER;

- **Return Parameters**

ER	ercd	Terminated normally (E_OK)

[[Assembly language API]]

```
.include mr8c.inc
ref_ver   PK_VER
```

- **Parameters**

PK_VER	Pointer to the packet to which version information is returned

- **Register contents after service call is issued**

Register name	Content after service call is issued
R0	Error code
A0	Pointer to the packet to which version information is returned

[[Error code]]

None

[[Functional description]]

This service call reads out information about the version of the currently executing kernel and returns the result to the area pointed to by pk_rver.

The following information is returned to the packet pointed to by pk_rver.

- **maker**
  The code H'115 denoting Renesas Technology Corporation is returned.
- **prid**
  The internal identification code IDH'0016 of the MR8C/4 is returned.
- **spver**
  The code H'5402 denoting that the kernel is compliant with µITRON Specification Ver 4.02.00 is returned.
- **prver**
  The code H'0100 denoting the version of the MR8C/4 is returned.
- **prno**

prno[0]
Reserved for future extension.

prno[1]
Reserved for future extension.

prno[2]
Reserved for future extension.

prno[3]
Reserved for future extension.

This service call can only be issued from task context, and cannot be issued from non-task context.

[[Example program statement]]

<<Example statement in C language>>

```
  #include <itron.h>
  #include <kernel.h>
  #include "kernel_id.h"
  void task()
  {
      T_RVER   pk_rver;
      ref_ver( &pk_rver );
  }
```

<<Example statement in assembly language>>

```
     .include mr8c.inc
     .GLB        task
_ refver:        .blkb   6
task:
          :
        PUSHM   A0
        ref_ver #_refver
          :
```

6. Applications Development Procedure Overview

6.1 Overview

Application programs for MR8C/4 should generally be developed following the procedure described below.

1. Generating a project

When using High-performance Embedded Workshop, create a new project using MR8C/4 on High-performance Embedded Workshop.

2. Coding the application program

Write the application program in code form using C or assembly language. If necessary, correct the sample start-tup program (crt0mr.a30) and section definition file (c_sec.inc or asm_sec.inc).

3. Creating a configuration file

Create a configuration file which has defined in it the task entry address, stack size, etc. by using an editor.

The GUI configurator available for MR8C/4 may be used to create a configuration file.

4. Executing the configurator

From the configuration file, create system data definition files (sys_rom.inc, sys_ram.inc) and include files (mr8c.inc, kernel_id.h, kernel_sysint.h).

5. System generation

Execute the make command or execute build on High-performance Embedded Workshop to generate a system.

6. Writing to ROM

Using the ROM programming format file created, write the finished program file into the ROM. Or load it into the debugger to debug.

Figure 6.1 shows a detailed flow of system generation.

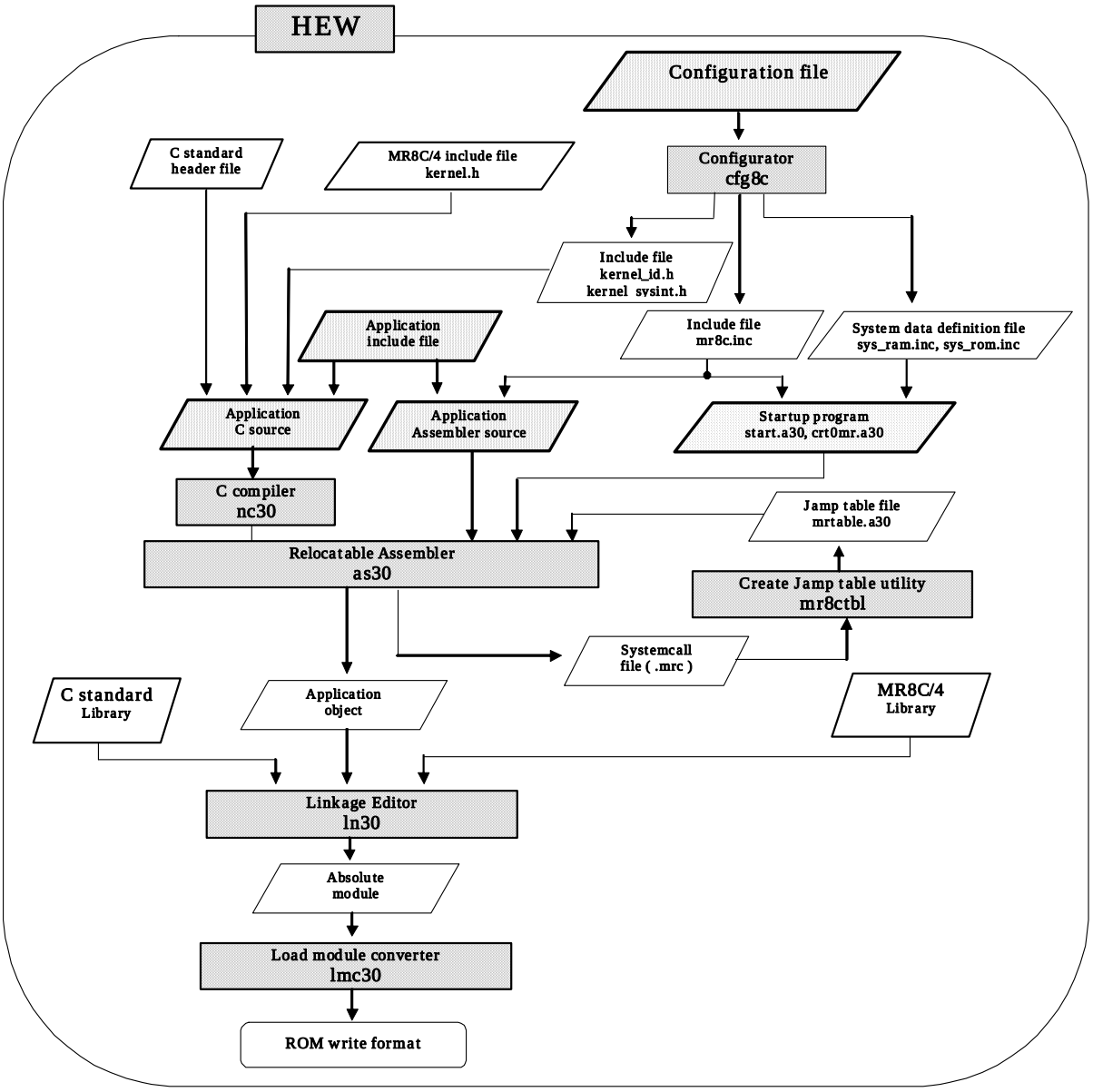


Figure 6.1 MR8C/4 System Generation Detail Flowchart

7. Detailed Applications

7.1 Program Coding Procedure in C Language

7.1.1 Task Description Procedure

1. Describe the task as a function.

To register the task for the MR8C/4, enter its function name in the configuration file. When, for instance, the function name "task()" is to be registered as the task ID number 3, proceed as follows.

```
task[3]{
        name            = ID_task;

        entry_address   = task();

        stack_size      = 100;

        priority        = 3;

};
```

2. At the beginning of file, be sure to include "itron.h",”kernel.h” which is in system directory as well as "kernel_id.h" which is in the current directory. That is, be sure to enter the following two lines at the beginning of file.

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"
```

3. No return value is provided for the task start function. Therefore, declare the task start function as a void function.

4. A function that is declared to be static cannot be registered as a task.

5. It isn't necessary to describe ext_tsk() at the exit of task start function.[23]If you exit the task from the subroutine in task start function, please describe ext_tsk() in the subroutine.

6. It is also possible to describe the task startup function, using the infinite loop.

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void task(void)
{

        /* process */

}
```

Figure 7.1 Example Infinite Loop Task Described in C Language

[23] The task is ended by ext_tsk() automatically if #pramga TASK is declared in the MR8C/4. Similarly, it is ended by ext_tsk when returned halfway of the function by return sentence.

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void task(void)
{
        for(;;){
                /* process */

        }
}
```

Figure 7.2 Example Task Terminating with ext_tsk() Described in C Language

7. **To specify a task, use the string written in the task definition item "name" of the configuration file.[24]**

```
wup_tsk(ID_main);
```

8. **To specify an event flag,semaphore, or data queue, use the respective strings defined in the configuration file.**

   For example, if an event flag is defined in the configuration file as shown below,

```
flag[1]{
        name    = ID_abc;
};
```

   To designate this eventflag, proceed as follows.

```
set_flg(ID_abc,&setptn);
```

9. **To specify a cyclic or alarm handler, use the string written in the cyclic or alarm handler definition item "name" of the configuration file.**

```
sta_cyc(ID_cyc);
```

10. **When a task is reactivated by the sta_tsk() service call after it has been terminated by the ter_tsk() service call, the task itself starts from its initial state.[25] However, the external variable and static variable are not automatically initialized when the task is started. The external and static variables are initialized only by the startup program (crt0mr.a30), which actuates before MR8C/4 startup.**

11. **The task executed when the MR8C/4 system starts up is setup.**

12. **The variable storage classification is described below.**

    The MR8C/4 treats the C language variables as indicated in Table 7.1 C Language Variable Treatment.

Table 7.1 C Language Variable Treatment

Variable storage class	Treatment
Global Variable	Variable shared by all tasks
Non-function static variable	Variable shared by the tasks in the same file
Auto Variable Register Variable Static variable in function	Variable for specific task

7.1.2 Writing a Kernel (OS Dependent) Interrupt Handler

When describing the kernel (OS-dependent) interrupt handler in C language, observe the following precautions.

[24] The configurator generates the file "kernel_id.h" that is used to convert the ID number of a task into the string to be specified. This means that the #define declaration necessary to convert the string specified in the task definition item "name" into the ID number of the task is made in "kernel_id.h." The same applies to the cyclic and alarm handlers.

[25] The task starts from its start function with the initial priority in a wakeup counter cleared state.

1. **Describe the kernel(OS-dependent) interrupt handler as a function [26]**
2. **Be sure to use the void type to declare the interrupt handler start function return value and argument.**
3. **At the beginning of file, be sure to include "itron.h","kernel.h" which is in the system directory as well as "kernel_id.h" which is in the current directory.**
4. **Do not use the ret_int service call in the interrupt handler.[27]**
5. **The static declared functions can not be registered as an interrupt handler.**

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void inthand(void)

{
        /* process */

        iwup_tsk(ID_main);
}
```

Figure 7.3 Example of Kernel(OS-dependent) Interrupt Handler

7.1.3 Writing Non-kernel (OS-independent) Interrupt Handler

When describing the non-kernel(OS-independent) interrupt handler in C language, observe the following precautions.

1. **Be sure to declare the return value and argument of the interrupt handler start function as a void type.**
2. **No service call can be issued from a non-kernel(an OS-independent) interrupt handler.**
   NOTE: If this restriction is not observed, the software may malfunction.
3. **A function that is declared to be static cannot be registered as an interrupt handler.**
4. **If you want multiple interrupts to be enabled in a non-kernel(an OS-independent) interrupt handler, always make sure that the non-kernel(OS-independent) interrupt handler is assigned a priority level higher than other kernel(OS-dependent) interrupt handlers.[28]**

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void inthand(void)
{
        /* process */
}
```

Figure 7.4 Example of Non-kernel(OS-independent) Interrupt Handler

7.1.4 Writing Cyclic Handler/Alarm Handler

When describing the cyclic or alarm handler in C language, observe the following precautions.

[26] A configuration file is used to define the relationship between handlers and functions.
[27] When an kernel(OS-dependent) interrupt handler is declared with #pragma INTHANDLER ,code for the ret_int service call is automatically generated.
[28] If you want the non-kernel(OS-independent) interrupt handler to be assigned a priority level lower than kernel(OS-dependent) interrupt handlers, change the description of the non-kernel(OS-independent) interrupt handler to that of the kernel (OS-dependent) interrupt handler.

1. **Describe the cyclic or alarm handler as a function.**[29]
2. **Be sure to declare the return value and argument of the interrupt handler start function as a void type.**
3. **At the beginning of file, be sure to include "itron.h",”kernel.h” which is in the system directory as well as "kernel_id.h" which is in the current directory.**
4. **The static declared functions cannot be registered as a cyclic handler or alarm handler.**
5. **The cyclic handler and alarm handler are invoked by a subroutine call from a system clock interrupt handler.**

```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void cychand(void)
{
        /*process */
}
```

Figure 7.5 Example Cyclic Handler Written in C Language

[29] The handler-to-function name correlation is determined by the configuration file.

7.2 Program Coding Procedure in Assembly Language

This section describes how to write an application using the assembly language.

7.2.1 Writing Task

This section describes how to write an application using the assembly language.

1. **Be sure to include "mr8c.inc" at the beginning of file.**
2. **For the symbol indicating the task start address, make the external declaration.[30]**
3. **Be sure that an infinite loop is formed for the task or the task is terminated by the ext_tsk service call.**

```
        .INCLUDE mr8c.inc ----- (1)
        .GLB    task    ----- (2)

task:
                ; process
        jmp   task      ----- (3)
```

Figure 7.6 Example Infinite Loop Task Described in Assembly Language

```
        .INCLUDE mr8c.inc
        .GLB    task

task:
                ; process
        ext_tsk
```

Figure 7.7 Example Task Terminating with ext_tsk Described in Assembly Language

4. **The initial register values at task startup are indeterminate except the PC, SB, R0 and FLG registers.**
5. **To specify a task, use the string written in the task definition item “name” of the configuration file.**

```
wup_tsk  #ID_task
```

6. **To specify an event flag, semaphore, or data queue, use the respective strings defined in the configuration file.**

   For example, if a semaphore is defined in the configuration file as shown below,:

```
semaphore[1]{
        name            = abc;
};
```

   To specify this semaphore, write your specification as follows:

```
sig_sem  #ID_abc
```

7. **To specify a cyclic or alarm handler, use the string written in the cyclic or alarm handler definition item “name” of the configuration file**

   For example, if you want to specify a cyclic handler "cyc," write your specification as follows:

```
sta_cyc  #ID_cyc
```

[30] Use the .GLB pseudo-directive

8. **Set a task that is activated at MR8C/4 system startup in the configuration file [31]**

7.2.2 Writing Kernel(OS-dependent) Interrupt Handler

When describing the kernel(OS-dependent) interrupt handler in assembly language, observe the following precautions

1. **At the beginning of file, be sure to include "mr8c.inc" which is in the system directory.**
2. **For the symbol indicating the interrupt handler start address, make the external declaration(Global declaration).[32]**
3. **Make sure that the registers used in a handler are saved at the entry and are restored after use.**
4. **Return to the task by ret_int service call.**

```
        .INCLUDE mr8c.inc               ------(1)
        .GLB    inth                    ------(2)

inth:
        ;  Registers used are saved to a stack   ------(3)
        iwup_tsk  #ID_task1
                :
                process
                :

        ;  Registers used are restored  ------(3)

        ret_int                         ------(4)
```

Figure 7.8 Example of kernel(OS-depend) interrupt handler

7.2.3 Writing Non-kernel(OS-independent) Interrupt Handler

1. **For the symbol indicating the interrupt handler start address, make the external declaration (public declaration).**
2. **Make sure that the registers used in a handler are saved at the entry and are restored after use.**
3. **Be sure to end the handler by REIT instruction.**
4. **No service calls can be issued from a non-kernel(an OS-independent) interrupt handler.**
   NOTE: If this restriction is not observed, the software may malfunction.
5. **If you want multiple interrupts to be enabled in a non-kernel(an OS-independent) interrupt handler, always make sure that the non-kernel(OS-independent) interrupt handler is assigned a priority level higher than other non-kernel(OS-dependent) interrupt handlers.[33]**

```
        .GLB    inthand                 ----- (1)

inthand:
        ; Registers used are saved to a stack    ----- (2)
        ; interrupt process
        ; Registers used are restored   ----- (2)
        REIT                            ----- (3)
```

Figure 7.9 Example of Non-kernel(OS-independent) Interrupt Handler of Specific Level

7.2.4 Writing Cyclic Handler/Alarm Handler

When describing the cyclic or alarm handler in Assembly Language, observe the following precautions.

[31] The relationship between task ID numbers and tasks(program) is defined in the configuration file.
[32] Use the .GLB peudo-directive.
[33] If you want the non-kernel(OS-independent) interrupt handler to be assigned a priority level lower than kernel(OS-dependent) interrupt handlers, change the description of the non-kernel(OS-independent) interrupt handler to that of the kernel (OS-dependent) interrupt handler.

1. **At the beginning of file, be sure to include "mr8c.inc" which is in the system directory.**
2. **For the symbol indicating the handler start address, make the external declaration.[34]**
3. **Always use the RTS instruction (subroutine return instruction) to return from cyclic handlers and alarm handlers.**

   For examples:

```
.INCLUDE        mr8c.inc        ----- (1)
.GLB            cychand         ----- (2)

cychand:
                :
        ; handler process
                :

        rts                     ----- (3)
```

Figure 7.10 Example Handler Written in Assembly Language

[34] Use the .GLB pseudo-directive.

7.3 Modifying MR8C/4 Startup Program

MR8C/4 comes with two types of startup programs as described below.

- start.a30
  This startup program is used when you created a program using the assembly language.

- crt0mr.a30
  This startup program is used when you created a program using the C language.

  This program is derived from "start.a30" by adding an initialization routine in C language.

The startup programs perform the following:

- Initialize the processor after a reset.
- Initialize C language variables (crt0mr.a30 only).
- Set the system timer.
- Initialize MR8C/4's data area.

Copy these startup programs from the directory indicated by environment variable "LIB8C" to the current directory.

If necessary, correct or add the sections below:

- Setting processor mode register
  Set a processor mode matched to your system to the processor mode register. (53th line in crt0mr.a30)

- Adding user-required initialization program
  When there is an initialization program that is required for your application, add it to the 140th line in the C language startup program (crt0mr.a30).

- Initialization of the standard I/O function
  Comment out the 96th – 97th line in the C language startup program (crt0mr.a30) if no standard I/O function is used.

7.3.1 C Language Startup Program (crt0mr.a30)

Figure 7.11 shows the C language startup program(crt0mr.a30).

```
; ***************************************************************
;
;    MR8C/4 start up program for C language
;    COPYRIGHT(C) 2009 RENESAS TECHNOLOGY CORPORATION
;    AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;
; ***************************************************************
; $Id: crt0mr.a30 519 2006-04-24 13:36:30Z inui $
;
    .list   OFF
    .include        c_sec.inc
    .include        mr8c.inc
    .include        sys_rom.inc
    .include        sys_ram.inc
    .list   ON

;------------------------------------------------------------------------
; SBDATA area definition
;------------------------------------------------------------------------
    .glb    __SB__
    .SB     __SB__

;=========================================================================
; Initialize Macro declaration
;------------------------------------------------------------------------
N_BZERO     .macro  TOP_,SECT_
    mov.b   #00H, R0L
    mov.w   #(TOP_ & 0FFFFH), A1
    mov.w   #sizeof SECT_, R3
    sstr.b
    .endm

N_BCOPY .macro      FROM_,TO_,SECT_
    mov.w   #(FROM_ & 0FFFFH),A0
    mov.b   #(FROM_>>16),R1H
    mov.w   #TO_,A1
    mov.w   #sizeof SECT_, R3
    smovf.b
    .endm

;=========================================================================
; Interrupt section start
;------------------------------------------------------------------------
    .glb    __SYS_INITIAL
    .section        MR_KERNEL,CODE,ALIGN
__SYS_INITIAL:
;------------------------------------------------------------------------
; after reset,this program will start
;------------------------------------------------------------------------
    ldc     #(__Sys_Sp&0FFFFH),ISP  ; set initial ISP

    mov.b   #2H,0AH
    mov.b   #00,PMOD                ; Set Processor Mode Regsiter
    mov.b   #0H,0AH
    ldc     #00H,FLG
    ldc     #(__Sys_Sp&0FFFFH),fb
    ldc     #__SB__,sb

; +-----------------------------------------------------+
; |    ISSUE SYSTEM CALL DATA INITIALIZE              |
; +-----------------------------------------------------+
    ; For PD30
    __INIT_ISSUE_SYSCALL

; +-----------------------------------------------------+
; |    MR RAM DATA 0(zero) clear              |
; +-----------------------------------------------------+
    N_BZERO MR_RAM_top,MR_RAM


;=========================================================================
; NEAR area initialize.
;------------------------------------------------------------------------
; bss zero clear
;------------------------------------------------------------------------
    N_BZERO bss_SE_top,bss_SE
```

```
 77     N_BZERO bss_SO_top,bss_SO
 78
 79     N_BZERO bss_NE_top,bss_NE
 80     N_BZERO bss_NO_top,bss_NO
 81
 82 ;---------------------------------------------------------------------
 83 ; initialize data section
 84 ;---------------------------------------------------------------------
 85     N_BCOPY data_SEI_top,data_SE_top,data_SE
 86     N_BCOPY data_SOI_top,data_SO_top,data_SO
 87     N_BCOPY data_NEI_top,data_NE_top,data_NE
 88     N_BCOPY data_NOI_top,data_NO_top,data_NO
 89
 90     ldc     #(__Sys_Sp&0FFFFH),     sp
 91     ldc     #(__Sys_Sp&0FFFFH),     fb
 92
 93 ;=====================================================================
 94 ; Initialize standard I/O
 95 ;---------------------------------------------------------------------
 96 ;    .glb    __init
 97 ;    jsr.a   __init
 98
 99 ;---------------------------------------------------------------------
100 ; Set System IPL
101 ; and
102 ; Set Interrupt Vector
103 ;---------------------------------------------------------------------
104     mov.b   #0,R0L
105     mov.b   #__SYS_IPL,R0H
106     ldc     R0,FLG                  ; set system IPL
107     ldc     #((__INT_VECTOR>>16)&0FFFFH),INTBH
108     ldc     #(__INT_VECTOR&0FFFFH),INTBL
109
110 .IF USE_TIMER
111 ;+-----------------------------------------------------+
112 ;|     System timer interrupt setting                  |
113 ;+-----------------------------------------------------+
114     mov.b   #stmr_mod_val,stmr_mod_reg      ;set timer mode
115     mov.b   #stmr_int_IPL,stmr_int_reg      ;set timer IPL
116     mov.b   #stmr_cnt_lower,stmr_ctr_reg    ;set interval count
117     mov.b   #stmr_cnt_upper,stmr_pre_reg    ;set interval count
118     mov.b   #0, stmr_ioc_reg
119 .IF USE_TIMER_RB
120     mov.b   #0, stmr_ct2_reg
121     mov.b   #0, stmr_one_reg
122 .ENDIF
123     or.b    #stmr_bit+1,stmr_start          ;system timer start
124 .ENDIF
125
126 ; +----------------------------------------------------+
127 ; |     System timer initialize                       |
128 ; +----------------------------------------------------+
129 .IF     USE_SYSTEM_TIME
130       MOV.W   #__D_Sys_TIME_L, __Sys_time+4
131       MOV.W   #__D_Sys_TIME_M, __Sys_time+2
132       MOV.W   #__D_Sys_TIME_H, __Sys_time
133 .ENDIF
134
135 ; +----------------------------------------------------+
136 ; |     User Initial Routine ( if there are )          |
137 ; +----------------------------------------------------+
138 ;
139
140
141 ;   jmp     __MR_INIT       ; for Separate ROM
142
143 ; +----------------------------------------------------+
144 ; |     Initalization of System Data Area              |
145 ; +----------------------------------------------------+
146     .GLB    __init_sys,__init_tsk,__END_INIT
147     JSR.W   __init_sys
148     JSR.W   __init_tsk
149
150    .IF      __NUM_FLG
151     .GLB    __init_flg
152     JSR.W   __init_flg
153    .ENDIF
154
155    .IF      __NUM_SEM
156     .GLB    __init_sem
```

```
157     JSR.W   __init_sem
158    .ENDIF
159
160    .IF      __NUM_DTQ
161     .GLB    __init_dtq
162     JSR.W   __init_dtq
163    .ENDIF
164
165    .IF      ALARM_HANDLER
166     .GLB    __init_alh
167     JSR.W   __init_alh
168    .ENDIF
169
170    .IF      CYCLIC_HANDLER
171     .GLB    __init_cyh
172     JSR.W   __init_cyh
173    .ENDIF
174
175     ; For PD30
176     __LAST_INITIAL
177
178 __END_INIT:
179 ; +-----------------------------------------------------+
180 ; |    Start initial active task                        |
181 ; +-----------------------------------------------------+
182     __START_TASK
183
184     .glb    __rdyq_search
185     jmp.W   __rdyq_search
186
187 ; +-------------------------------------------+
188 ; |           Define Dummy                    |
189 ; +-------------------------------------------+
190     .glb    __SYS_DMY_INH
191 __SYS_DMY_INH:
192     reit
193
194 .IF  CUSTOM_SYS_END
195 ; +-------------------------------------------+
196 ; | Syscall exit rouitne to customize
197 ; +-------------------------------------------+
198     .GLB    __sys_end
199 __sys_end:
200     ; Customize here.
201     REIT
202 .ENDIF
203
204 ; +-------------------------------------------+
205 ; |           exit() function                 |
206 ; +-------------------------------------------+
207     .glb    _exit,$exit
208 _exit:
209 $exit:
210     jmp     _exit
211
212 ; +-------------------------------------------+
213 ; |           System down routine             |
214 ; +-------------------------------------------+
215     .GLB    __vsys_dwn
216 __vsys_dwn:
217     JMP.B   __vsys_dwn
218
219
220 .if USE_TIMER
221 ; +-------------------------------------------+
222 ; |      System clock interrupt handler       |
223 ; +-------------------------------------------+
224     .SECTION        MR_KERNEL,CODE,ALIGN
225     .glb            __SYS_STMR_INH, __SYS_TIMEOUT
226     .glb            __DBG_MODE,__SYS_ISS
227 __SYS_STMR_INH:
228     ; process issue system call
229     ; For PD30
230     __ISSUE_SYSCALL
231
232
233
234 ; System timer interrupt handler
235     _STMR_hdr
236     ret_int
```

```
237 .endif
238
239     .end
```

Figure 7.11 C Language Startup Program (crt0mr.a30)

The following explains the content of the C language startup program (crt0mr.a30).

1. **Incorporate a section definition file [11 in Figure 7.11]**
2. **Incorporate an include file for MR8C/4 [12 in Figure 7.11]**
3. **Incorporate a system ROM area definition file [13 in Figure 7.11]**
4. **Incorporate a system RAM area definition file [14 in Figure 7.11]**
5. **This is the initialization program __SYS_INITIAL that is activated immediately after a reset. [46 - 185 in Figure 7.11]**
   - Setting the System Stack pointer [50 in Figure 7.11]
   - Setting the processor mode register [52- 54 in Figure 7.11]
   - Setting the SB,FB register [55 - 57 in Figure 7.11]
   - Initial set the C language. [76 - 92 in Figure 7.11]
   - Setting OS interrupt disable level [104 - 106 in Figure 7.11]
   - Setting the address of interrupt vector table [107 and 108 in Figure 7.11]
   - Set MR8C/4's system clock interrupt [114 -124 in Figure 7.11]
   - Initialization of standard I/O function[96-97 in Figure 7.11]
     When using no standard input/output functions, remove the lines 96 and 97 in Figure 7.11.
   - Initial set MR8C/4's system timer [129-133 in Figure 7.11]
6. **Initial set parameters inherent in the application [140 in Figure 7.11]**
7. **Initialize the RAM data used by MR8C/4 [146 - 173 in Figure 7.11]**
8. **Sets the bit which shows the end of start-up processing[176 in Figure 7.11]**
9. **Activate the initial startup task. [182 in Figure 7.11]**
10. **This is a system clock interrupt handler [221-236 in Figure 7.11]**

7.4 Memory Allocation

This section describes how memory is allocated for the application program data.

Use the section file provided by MR8C/4 to set memory allocation.

MR8C/4 comes with the following two types of section files:

- asm_sec.inc
  This file is used when you developed your applications with the assembly language.

  Refer to 7.4.1 for details about each section.

- c_sec.inc
  This file is used when you developed your applications with the C language.

  c_sec.inc is derived from "asm_sec.inc" by adding sections generated by C compiler NC30.

  Refer to 7.4.2 for details about each section.

Modify the section allocation and start address settings in this file to suit your system.

The following shows how to modify the section file.

e.g.

```
If you want to change the program section start address from F0000H to F1000H
           .section        program
           .org    04000H  ; Correct this address to 05000H

                   ↓

           .section        program

           .org    05000H  ;
```

7.4.1 Section Allocation of start.a30

The section allocation of the sample startup program for the assembly language "start.a30" is defined in "asm_sec.inc".

Edit "asm_sec.inc" if section reallocation is required.

The following explains each section that is defined in the sample section definition file "asm_sec.inc".

- MR_RAM_DBG section
  This section is stored MR8C/4's debug function RAM data.

  This section must be mapped in the Internal RAM area.

- MR_RAM section
  This section is where the RAM data, MR8C/4's system management data, is stored that is referenced in absolute addressing.

  This section must be mapped in the Internal RAM area.

- stack section
  This section is provided for each task's user stack and system stack.

  This section must be mapped in the Internal RAM area.

- MR_KERNEL section
  This section is where the MR8C/4 kernel program is stored.

- MR_CIF section
  This section stores the MR8C/4 C language interface library.

- MR_ROM section
  This section stores data such as task start addresses that area referenced by the MR8C/4 kernel.

- program section
  This section stores user programs.

  This section is not used by the MR8C/4 kernel at all. Therefore, you can use this section as desired.

- INTERRUPT_VECTOR section

- FIX_INTERRUPT_VECTOR section
  This section stores interrupt vectors.

7.4.2 Section Allocation of crt0mr.a30

The section allocation of the sample startup program for the C language "crt0mr.a30" is defined in "c_sec.inc".

Edit "c_sec.inc" if section reallocation is required.

The sections defined in the sample section definition file "c_sec.inc" include the following sections that are defined in the section definition file "asm_sec.inc" of the sample startup program for the assembly language.

- data_SE section
- bss_SE section
- data_SO section
- bss_SO section
- data_NE section
- bss_NE section
- data_NO section
- bss_NO section
- rom_NE section
- rom_NO section
- data_SEI section
- data_SOI section
- data_NEI section
- data_NOI section

  These sections are those that are generated by NC30. These sections are not defined in the section file for the assembly language.

  Refer to the NC30 manual for details.

The diagram below shows the section allocation in the sample startup program. (See Figure 7.12 Selection Allocation in C Language Startup Program)

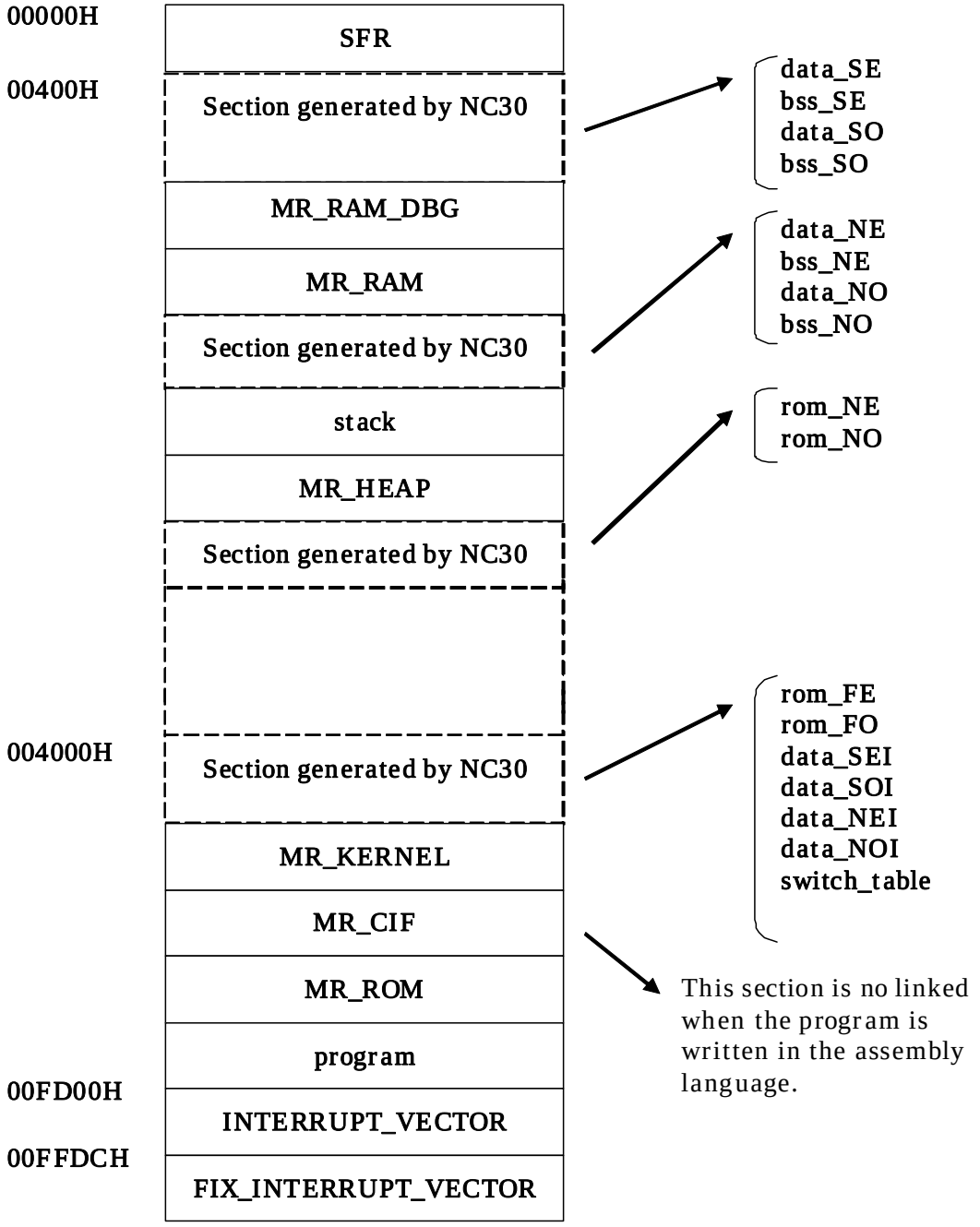


Figure 7.12 Selection Allocation in C Language Startup Program

8. Using Configurator

8.1 Configuration File Creation Procedure

When applications program coding and startup program modification are completed, it is then necessary to register the applications program in the MR8C/4 system.

This registration is accomplished by the configuration file.

8.1.1 Configuration File Data Entry Format

This chapter describes how the definition data are entered in the configuration file.

Comment Statement

A statement from '//' to the end of a line is assumed to be a comment and not operated on.

End of statement

Statements are terminated by ';'.

Numerical Value

Numerical values can be entered in the following format.

1. Hexadecimal Number

Add "0x" or "0X" to the beginning of a numerical value, or "h" or "H" to the end. If the value begins with an alphabetical letter between A and F with "h" or "H" attached to the end, be sure to add "0" to the beginning. Note that the system does not distinguish between the upper- and lower-case alphabetical characters (A-F) used as numerical values.[35]

2. Decimal Number

Use an integer only as in '23'. However, it must not begin with '0'.

3. Octal Numbers

Add '0' to the beginning of a numerical value of 'O' or 'o' to end.

4. Binary Numbers

Add 'B' or 'b' to the end of a numerical value. It must not begin with '0'.

Table 8.1 Numerical Value Entry Examples

Hexadecimal	0xf12 0Xf12 0a12h 0a12H 12h 12H
Decimal	32
Octal	017 17o 17O
Binary	101110b 101010B

[35] The system distinguishes between the upper- and lower-case letters except for the numbers A-F and a-f.

It is also possible to enter operators in numerical values. Table 8.2 Operators lists the operators available.

Table 8.2 Operators

Operator	Priority	Direction of computation
()	High	From left to right
- (Unary_minus)		From right to left
* / %		From left to right
+ - (Binary_minus)	Low	From loft to right

Numerical value examples are presented below.

- 123
- 123 + 0x23
- (23/4 + 3) * 2
- 100B + 0aH

Symbol

The symbols are indicated by a character string that consists of numerals, upper- and lower-case alphabetical letters, _(underscore), and ?, and begins with a non-numeric character.

Example symbols are presented below.

- _TASK1
- IDLE3

Function Name

The function names are indicated by a character string that consists of numerals, upper and lower-case alphabetical letters,'$'(dollar) and '_'(underscore), begins with a non-numeric character, and ends with '()'.

The following shows an example of a function name written in the C language.

- main()
- func()

When written in the assembly language, the start label of a module is assumed to be a function name.

Frequency

The frequency is indicated by a character string that consist of numerals and . (period), and ends with MHz. The numerical values are significant up to six decimal places. Also note that the frequency can be entered using decimal numbers only.

Frequency entry examples are presented below.

- 16MHz
- 8.1234MHz

It is also well to remember that the frequency must not begin with . (period).

8.1.2 Configuration File Definition Items

The following definitions [36] are to be formulated in the configuration file

- System definition
- System clock definition
- Task definition
- Eventflag definition
- Semaphore definition
- Data queue definition
- Cyclic handler definition
- Alarm handler definition
- Interrupt vector definition

[(System Definition Procedure)]

<< Format >>

```
// System Definition
system{
    stack_size      = System stack size ;
    priority        = Maximum value of priority ;
    system_IPL      = Kernel mask level(OS interrupt disable level) ;
    tic_deno        = Time tick denominator ;
    tic_nume        = Time tick numerator ;
};
```

<< Content >>

1. **System stack size**

   [(Definition format)] Numeric value

   [(Definition range)] 4 to 0xFFFF

   [(Default value)] 400H

   Define the total stack size used in service call and interrupt processing.

2. **Maximum value of priority (value of lowest priority)**

   [(Definition format)] Numeric value

   [(Definition range)] 1 to 255

   [(Default value)] 63

   Define the maximum value of priority used in MR8C/4's application programs. This must be the value of the highest priority used.

[36] All items except task definition can omitted. If omitted, definitions in the default configuration file are referenced.

3. Kernel mask level (OS interrupt disable level)

[(Definition format)] Numeric value

[(Definition range)] 1 to 7

[(Default value)] 7

Set the IPL value in service calls, that is, the OS interrupt disable level.

4. Time tick denominator

[(Definition format)] Numeric value

[(Definition range)] Fixed to 1

[(Default value)] 1

Set the denominator of the time tick.

5. Time tick numerator

[(Definition format)] Numeric value

[(Definition range)] 1 to 65,535

[(Default value)] 1

Set the numerator of the time tick. The system clock interrupt interval is determined by the time tick denominator and numerator that are set here. The interval is the time tick numerator divided by time tick denominator [ms]. That is, the time tick numerator [ms].

[(System Clock Definition Procedure)]

<< **Format** >>

```
// System Clock Definition
clock{
    mpu_clock       = MPU clock ;
    timer           = Timers used for system clock ;
    IPL             = System clock interrupt priority level ;
};
```

<< **Content** >>

1. MPU clock

[(Definition format)] Frequency(in MHz)

[(Definition range)] None

[(Default value)] 20MHz

Define the MPU operating clock frequency of the microcomputer in MHz units.

2. Timers used for system clock

[(Definition format)] Symbol

[(Definition range)] RA, RB, OTHER, NOTIMER

[(Default value)] NOTIMER

Define the hardware timers used for the system clock.

If you do not use a system clock, define "NOTIMER."

1. Initialize the timer in start-up routine

The cfg8c outputs following macros to the "kernel_id.h". Please initialize the timer based on this information.

__MR_MPUCLOCK	MPU operating frequency described in the cfg file.
__MR_UNITTIME	The interrupt interval of the system clock was expressed with us.
__MR_TIMER_IPL	The IPL value of the system clock interrupt

2. Define the relocatable interrupt vector as follows.

```
interrupt_vector[<Vector number>] {
      entry_address =    __RI_SYS_STMR_INH;
      os_int = YES;
};
```

3. System clock interrupt priority level

[(Definition format)] Numeric value

[(Definition range)] 1 to Kernel mask(OS interrupt disable) level in system definition

[(Default value)] 4

Define the priority level of the system clock timer interrupt. The value set here must be smaller than the kernel mask(OS interrupt disable level.

Interrupts whose priority levels are below the interrupt level defined here are not accepted during system clock interrupt handler processing.

[(Task definition)]

<< Format >>

```
 // Tasks Definition
task[ ID No. ]{
    name            = ID name;
    entry_address   = Start task of address;
    stack_size      = User stack size of task;
    priority        = Initial priority of task;
    context         = Registers used;
    stack_section   = Section name in which the stack is located;
    initial_start   = TA ACT attribute (initial startup state);
    exinf           = Extended information;
};
            :
            :
```

The ID number must be in the range of 1 to 255. The ID number can be omitted.

If omitted, numbers are automatically assigned sequentially beginning with the smallest.

<< **Content** >>

Define the following for each task ID number.

1. Task ID name

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] None

Define the ID name of a task. Note that the function name defined here is output to the kernel_id.h file, as shown below.

#define Task ID Name task ID

2. Start address of task

[(Definition format)] Symbol or function name

[(Definition range)] None

[(Default value)] None

Define the entry address of a task. When written in the C language, add () at the end or _at the beginning of the function name you have defined.

The function name defined here causes the following declaration statement to be output in the kernel_id.h file:

```
#pragma TASK Function Name
```

3. User stack size of task

[(Definition format)] Numeric value

[(Definition range)] 6 or more

[(Default value)] 256

Define the user stack size for each task. The user stack means a stack area used by each individual task. MR8C/4 requires that a user stack area be allocated for each task, which amount to at least 12 bytes.

4. Initial priority of task

[(Definition format)] Numeric value

[(Definition range)] 1 to (maximum value of priority in system definition)

[(Default value)] 1

Define the priority of a task at startup time.

As for MR8C/4's priority, the lower the value, the higher the priority.

5. Regisers Used

[(Definition format)] Symbol[,Symbol,....]

[(Definition range)] Selected from R0,R1,R2,R3,A0,A1,SB,FB

[(Default value)] All registers

Define the registers used in a task. MR8C/4 handles the register defined here as a context. Specify the R0 and R1 register because task startup code is set in R1 when the task starts and return parameter is returned in R0.

However, the registers used can only be selected when the task is written in the assembly language. Select all registers when the task is written in the C language. When selecting a register here, be sure to select all registers that store service call parameters used in each task.

MR8C/4 kernel does not change the registers of bank.

If this definition is omitted, it is assumed that all registers are selected.

6. Section name in which the stack is located

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] stack

Define the section name in which the stack is located. The section defined here must always have an area allocated for it in the section file (asm_sec.inc or c_sec.inc).

If no section names are defined, the stack is located in the stack section.

7. TA_ACT attribute (initial startup state)

[(Definition format)] Symbol

[(Definition range)] ON or OFF

[(Default value)] OFF

Define the initial startup state of a task.

If this attribute is specified ON, the task goes to a READY state at the initial system startup time.

The task startup code of the initial startup task is 0. One or more tasks must have TA_ACT attribute.

8. Extended information

[(Definition format)] Numeric value

[(Definition range)] 0 to 0xFFFF

[(Default value)] 0

Define the extended information of a task. This information is passed to the task as argument when it is restarted by a queued startup request, for example.

[(Eventflag definition)]

This definition is necessary to use Eventflag function.

<< Format >>

```
// Eventflag Definition
flag[ ID No. ]{
    name            = Name ;
    initial_pattern = Initial value of the event flag ;
    wait_multi      = Multi-wait attribute ;
    clear_attribute = Clear attribute ;
};
            :
            :
```

The ID number must be in the range of 1 to 255. The ID number can be omitted.

If omitted, numbers are automatically assigned sequentially beginning with the smallest.

<< Content >>

Define the following for each eventflag ID number.

1. ID Name

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] None

Define the name with which an eventflag is specified in a program.

2. Initial value of the event flag

[(Definition format)] Numeric value

[(Definition range)] 0 to 0xFFFF

[(Default value)] 0

Specify the initial bit pattern of the event flag.

3. Multi-wait attribute

[(Definition format)] Symbol

[(Definition range)] TA_WMUL or TA_WSGL

[(Default value)] TA_WSGL

Specify whether multiple tasks can be enqueued in the eventflag waiting queue. If TA_WMUL is selected, the TA_WMUL attribute is added, permitting multiple tasks to be enqueued. If TA_WSGL is selected, the TA_WSGL attribute is added, prohibiting multiple tasks from being enqueued.

4. Clear attribute

[(Definition format)] Symbol

[(Definition range)] YES or NO

[(Default value)] NO

Specify whether the TA_CLR attribute should be added as an eventflag attribute. If YES is selected, the TA_CLR attribute is added. If NO is selected, the TA_CLR attribute is not added.

[(Semaphore definition)]

This definition is necessary to use Semaphore function.

<< Format >>

```
// Semaphore Definition
semaphore[ ID No. ]{
    name            = ID name;
    initial_count   = Initial value of semaphore counter;
    max_count       = Maximum value of the semaphore counter;
};
            :
            :
```

The ID number must be in the range of 1 to 255. The ID number can be omitted.

If omitted, numbers are automatically assigned sequentially beginning with the smallest.

<< **Content** >>

Define the following for each semaphore ID number.

1. ID Name

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] None

Define the name with which a semaphore is specified in a program.

2. Initial value of semaphore counter

[(Definition format)] Numeric value

[(Definition range)] 0 to 65535

[(Default value)] 1

Define the initial value of the semaphore counter.

3. Maximum value of the semaphore counter

[(Definition format)] Numeric value

[(Definition range)] 1 to 65535

[(Default value)] 1

Define the maximum value of the semaphore counter.

[(Data queue definition)]

This definition must always be set when the data queue function is to be used.

<< **Format** >>

```
// Dataqueue Definition
dataqueue[ ID No. ]{
    name            = ID name;
    buffer_size     = Number of data queues;
};
            :
            :
```

The ID number must be in the range 1 to 255. The ID number can be omitted. If omitted, ID numbers are automatically assigned in order of numbers beginning with the smallest.

<< **Content** >>

For each data queue ID number, define the items described below.

1. ID name

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] None

Define the name by which the data queue is specified in a program.

2. Number of data

[(Definition format)] Numeric Value

[(Definition range)] 0 to 0x3FFF

[(Default value)] 0

Specify the number of data that can be transmitted. What should be specified here is the number of data, and not a data size.

[(Cyclic handler definition)]

This definition is necessary to use Cyclic handler function.

<< Format >>

```
// Cyclic Handlar Definition
 cyclic_hand[ ID No. ]{
    name                    = ID name ;
    interval_counter        = Activation cycle ;
    start                   = TA_STA attribute ;
    phsatr                  = TA_PHS attribute ;
    phs_counter             = Activation phase ;
    entry_address           = Start address;
    exitf                   = Extended information ;
 };
            :
            :
```

The ID number must be in the range of 1 to 255. The ID number can be omitted.

If omitted, numbers are automatically assigned sequentially beginning with the smallest.

<< Content >>

Define the following for each cyclic handler ID number.

1. ID name

[(Definition format)] Symbol

[(Definition range)] None

[(Default value)] None

Define the name by which the cyclic handler is specified in a program.

2. Activation cycle

[(Definition format)] Numeric value

[(Definition range)] 1 to 0x7FFFFFFF

[(Default value)] None

Define the activation cycle at which time the cyclic handler is activated periodically. The activation cycle here must be defined in the same unit of time as the system clock's unit time that is defined in system clock definition item. If you want the cyclic handler to be activated at 1-second intervals, for example, the activation cycle here must be set to 1000.

3. TA_STA attribute

[(Definition format)] Symbol

[(Definition range)] ON or OFF

[(Default value)] OFF

Specify the TA_STA attribute of the cyclic handler. If ON is selected, the TA_STA attribute is added; if OFF is selected, the TA_STA attribute is not added.

4. TA_PHS attribute

[(Definition format)] Symbol

[(Definition range)] ON or OFF

[(Default value)] OFF

Specify the TA_PHS attribute of the cyclic handler. If ON is selected, the TA_PHS attribute is added; if OFF is selected, the TA_PHS attribute is not added.

5. Activation phase

[(Definition format)] Numeric value

[(Definition range)] 0 to 0x7FFFFFFF

[(Default value)] None

Define the activation phase of the cyclic handler. The time representing this startup phase must be defined in ms units.

6. Start Address

[(Definition format)] Symbol or Function Name

[(Definition range)] None

[(Default value)] None

Define the start address of the cyclic handler.

Note that the function name defined here will have the declaration statement shown below output to the kernel_id.h file.

```
#pragma CYCHANDLER function name
```

7. Extended information

[(Definition format)] Numeric value

[(Definition range)] 0 to 0xFFFF

[(Default value)] 0

Define the extended information of the cyclic handler. This information is passed as argument to the cyclic handler when it starts.

[(Alarm handler definition)]
This definition is necessary to use Alarm handler function.

<< **Format** >>

```
// Alarm Handlar Definition
alarm_hand[ ID No. ]{
    name            = ID name;
    entry_address   = Start address;
    exitf           = Extended information ;
};
            :
            :
```

The ID number must be in the range of 1 to 255. The ID number can be omitted.

If omitted, numbers are automatically assigned sequentially beginning with the smallest.

<< **Content** >>

Define the following for each alarm handler ID number.

1. **ID name**

   [(Definition format)] Symbol

   [(Definition range)] None

   [(Default value)] None

   Define the name by which the alarm handler is specified in a program.

2. **Start address**

   [(Definition format)] Symbol or Function Name

   [(Definition range)] None

   Define the start address of the alarm handler. The function name defined here causes the following declaration statement to be output in the kernel_id.h file.

3. **Extended information**

   [(Definition format)] Numeric value

   [(Definition range)] 0 to 0xFFFF

   [(Default value)] 0

   Define the extended information of the alarm handler. This information is passed as argument to the alarm handler when it starts.

[(Interrupt vector definition)]
This definition is necessary to use Interrupt function.

<< Format >>

```
// Interrupt Vector Definition
interrupt_vector[ Vector No. ]{
   os_int          = Kernel-managed (OS dependent) interrupt handler;
   entry_address   = Start address;
   pragma_switch   = Switch passed to PRAGMA extended function;
};
            :
            :
```

The vector number can be written in the range of 0 to 63 and 247 to 255. However, whether or not the defined vector number is valid depends on the microcomputer used

Configurator can't create an Initialize routine (interrupt control register, interrupt causes etc.) for this defined interrupt. You need to create that.

<< Content >>

1. Kernel (OS dependent) interrupt handler

[(Definition format)] Symbol

[(Definition range)] YES or NO

Define whether the handler is a kernel(OS dependent) interrupt handler. If it is a kernel(OS dependent) interrupt handler, specify YES; if it is a non-kernel(OS independent) interrupt handler, specify No.

If this item is defined as YES, the declaration statement shown below is output to the kernel_id.h file.

```
#pragma INTHANDLER /V4 function name
```

If this item is defined as NO, the declaration statement shown below is output to the kernel_id.h file.

```
#pragma INTERRUPT /V4 function name
```

2. Start address

[(Definition format)] Symbol or function name

[(Definition range)] None

[(Default value)] __SYS_DMY_INH

Define the entry address of the interrupt handler. When written in the C language, add () at the end or at the beginning of the function name you have defined.

3. Switch passed to PRAGMA extended function

[(Definition format)] Symbol

[(Definition range)] E, F, or B

[(Default value)] None

Specify the switch to be passed to #pragma INTHANDLER or #pragma INTERRUPT. If "E" is specified, the "/E" switch is assumed, in which case multiple interrupts (another interrupt within an interrupt) are enabled. If "F" is specified, the "/F" switch is assumed, in which case the FREIT instruction is output at return from the interrupt handler. If "B" is specified, the "/B" switch is assumed, in which case register bank 1 is specified.

Two or more switches can be specified at the same time. For kernel (OS dependent) interrupt handlers, however, only the "E" switch can be specified. For non-kernel (OS independent) interrupt handlers, the "E," "F," and "B" switches can be specified, subject to a limitation that "E" and "B" cannot be specified at the same time.

[Precautions]

1. Regarding the method for specifying a register bank

A kernel (OS dependent) interrupt handler that uses register bank 1 cannot be written in C language. Such an interrupt handler can only be written in assembly language. When writing in assembly language, make sure the statements at the entry and exit of the interrupt handler are written as shown below.

(Always be sure to clear the B flag before issuing the ret_int service call.)

Example: interrupt;

```
        fset    B
        fclr    B
        ret_int
```

Internally in the MR8C/4 kernel, register banks are not switched over.

2. Regarding the method for specifying a high-speed interrupt

To ensure an effective use of high-speed interrupts, make sure the registers of register bank 1 are used in the high-speed interrupt. Note also that high-speed interrupts cannot be used for the kernel (OS dependent) interrupt handler.

3. Do not use watchdog timer interrupts in the kernel (OS dependent) interrupt.

The interrupt factors and the vector number of a fixed vector are shown as follows below. Please refer to the hardware manual of the microcomputer that is use for a changeable vector.

Table 8.3 Correspondence of fixed vector interrupt factor and vector number

Interrupt Factor	Vector number	Section Name
Undefined instruction	247	FIX_INTERRUPT_VECTOR
Overflow	248	FIX_INTERRUPT_VECTOR
BRK instruction	249	FIX_INTERRUPT_VECTOR
Addres-match	250	FIX_INTERRUPT_VECTOR
Single-step	251	FIX_INTERRUPT_VECTOR
Watchdog	252	FIX_INTERRUPT_VECTOR
Adress break	253	FIX_INTERRUPT_VECTOR
Reserved	254	FIX_INTERRUPT_VECTOR
Reset	255	FIX_INTERRUPT_VECTOR

8.1.3 Configuration File Example

The following is the configuration file example.

```
 1 //*********************************************************************
 2 //
 3 //   COPYRIGHT(C) 2009 RENESAS TECHNOLOGY CORPORATION
 4 //   AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
 5 //   MR8C/4 V.1.00
 6 //
 7 //   MR8C/4 System Configuration File.
 8 //
 9 //*********************************************************************
10 system{
11      stack_size      = 0x400;
12      priority        = 16;
13      system_IPL      = 7;
14      tic_deno        = 1;
15      tic_nume        = 1;
16 };
17 //System Clock Definition
18 clock{
19      mpu_clock       = 20MHz;
20      timer                   = RB;
21      IPL                     = 5;
22 };
23 //Task Definition
24 task[1]{
25      name                    = TASK_ID1;
26      initial_start           = ON;
27      entry_address           = task1();
28      stack_size              = 0x80;
29      priority                = 9;
30      exinf = 0x1234;
31 };
32 task[2]{
33      name                    = TASK_ID2;
34      initial_start           = OFF;
35      entry_address           = task2();
36      stack_size              = 0x80;
37      priority                = 2;
38      exinf = 0x8000;
39 };
40 task[3]{
41      name                    = TASK_ID3;
42      initial_start           = OFF;
43      entry_address           = task3();
44      stack_size              = 0x80;
45      priority                = 3;
46      exinf = 0x1234;
47 };
48 //event flag default added
49 flag[1] {
50      name = FLG_ID1;
51      initial_pattern         = 0x0000;
52      wait_multi = TA_WMUL;
53      clear_attribute         = YES;
54 };
55 semaphore[1]{
56      name = SEM_ID1;
57      initial_count   = 0;
58
59 };
60 interrupt_vector[22] {
61      os_int = YES;
62      entry_address = inth ();
63 };
64
65 //
66 // End of Configuration
67 //
```

8.2 Configurator Execution Procedures

8.2.1 Configurator Overview

The configurator is a tool that converts the contents defined in the configuration file into the assembly language include file, etc.Figure 8.1 outlines the operation of the configurator.

When used on High-performance Embedded Workshop, the configurator is automatically started, and an application program is built.

1. Executing the configurator requires the following input files:

- Configuration file (XXXX.cfg)
  This file contains description of the system's initial setup items. It is created in the current directory.

- Default configuration file (default.cfg)
  This file contains default values that are referenced when settings in the configuration file are omitted. This file is placed in the directory indicated by environment variable "LIB8C" or the current directory. If this file exists in both directories, the file in the current directory is prioritized over the other.

- include template file(mr8c.inc, sys_ram.inc, kernel_sysint.h)
  This file serves as the template file of include file "mr8c.inc", “kernel_sysint.h” and “sys_ram.inc”. It resides in the directory indicated by environment variable "LIB8C."

- MR8C/4 version file (version)
  This file contains description of MR8C/4's version. It resides in the directory indicated by environment variable "LIB8C." The configurator reads in this file and outputs MR8C/4's version information to the startup message.

2. When the configurator is executed, the files listed below are output.

Do not define user data in the files output by the configurator. Starting up the configurator after entering data definitions may result in the user defined data being lost.

- System data definition file (sys_rom.inc, sys_ram.inc)
  This file contains definition of system settings.

- Include file (mr8c.inc)
  This is an include file for the assembly language.

- ID number definition file(kernel_id.h)
  The ID numbers of kernel objects are defined

- Service call information file(kernel_sysint.h)
  This is the include file of the service call usage information.

.

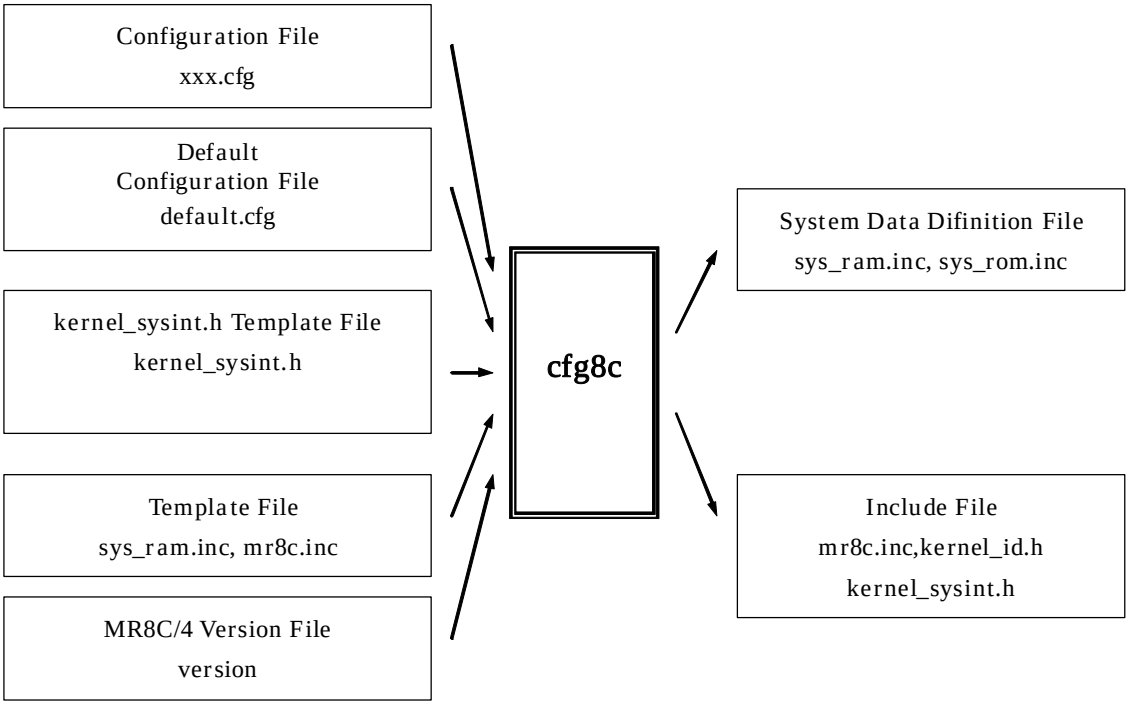


Figure 8.1 The operation of the Configurator

8.2.2 Setting Configurator Environment

Before executing the configurator, check to see if the environment variable "LIB8C" is set correctly.

The configurator cannot be executed normally unless the following files are present in the directory indicated by the environment variable "LIB8C":

- Default configuration file (default.cfg)

This file can be copied to the current directory for use. In this case, the file in the current directory is given priority.

- System RAM area definition database file (sys_ram.inc)
- mr8c.inc template file (mr8c.inc)
- Section definition file(c_sec.inc or asm_sec.inc)
- Startup file(crt0mr.a30 or start.a30)
- MR8C/4 version file(version)
- Service call information file(kernel_sysint.h)

8.2.3 Configurator Start Procedure

Start the configurator as indicated below.

```
C> cfg8c  [-vV]  Configuration file name
```

Normally, use the extension .cfg for the configuration file name.

Command Options

-v Option

Displays the command option descriptions and detailed information on the version.

-V Option

Displays the information on the files generated by the command.

Error messages

cfg8c Error : syntax error near line xxx (xxxx.cfg)

There is an syntax error in the configuration file.

cfg8c Error : not enough memory

Memory is insufficient.

cfg8c Error : illegal option --> <x>

The configurator's command option is erroneous.

cfg8c Error : illegal argument --> <xx>

The configurator's startup format is erroneous.

cfg8c Error : can't write open <XXXX>

The XXXX file cannot be created. Check the directory attribute and the remaining disk capacity available.

cfg8c Error : can't open <XXXX>

The XXXX file cannot be accessed. Check the attributes of the XXXX file and whether it actually exists.

cfg8c Error : can't open version file

The MR8C/4 version file "version" cannot be found in the directory indicated by the environment variable "LIB8C".

cfg8c Error : can't open default configuration file

The default configuration file cannot be accessed. "default.cfg" is needed in the current directory or directory "LIB8C" specifying.

cfg8c Error : can't open configuration file <xxxx.cfg>

The configuration file cannot be accessed. Check that the file name has been properly designated.

cfg8c Error : illegal XXXX --> <xx> near line xxx (xxxx.cfg)

The value or ID number in definition item XXXX is incorrect. Check the valid range of definition.

cfg8c Error : Unknown XXXX --> <xx> near line xx (xxxx.cfg)

The symbol definition in definition item XXXX is incorrect. Check the valid range of definition.

cfg8c Error : too big XXXX's ID number --> <xx> (xxxx.cfg)

A value is set to the ID number in XXXX definition that exceeds the total number of objects defined.The ID number must be smaller than the total number of objects.

cfg8c Error : too big task[x]'s priority --> <xx> near line xxx (xxxx.cfg)

The initial priority in task definition of ID number x exceeds the priority in system definition.

cfg8c Error : too big IPL --> <xx> near line xxx (xxxx.cfg)

The system clock interrupt priority level for system clock definition item exceeds the value of IPL within service call of system definition item.

cfg8c Error : system timer's vector <x>conflict near line xxx

A different vector is defined for the system clock timer interrupt vector. Confirm the vector No.x for interrupt vector definition.

cfg8c Error : XXXX is not defined (xxxx.cfg)

"XXXX" item must be set in your configuration file.

cfg8c Error : system's default is not defined

These items must be set int the default configuration file.

cfg8c Error : double definition <XXXX> near line xxx (xxx.cfg)

XXXX is already defined. Check and delete the extra definition.

cfg8c Error : double definition XXXX[x] near line xxx (default.cfg)
cfg8c Error : double definition XXXX[x] near line xxx (xxxx.cfg)

The ID number in item XXXX is already registered. Modify the ID number or delete the extra definition.

cfg8c Error : you must define XXXX near line xxx (xxxx.cfg)

XXXX cannot be omitted.

cfg8c Error : you must define SYMBOL near line xxx (xxxx.cfg)

This symbol cannot be omitted.

cfg8c Error : start-up-file (XXXX) not found

The start-up-file XXXX cannot be found in the current directory. The startup file "start.a30" or "crt0mr.a30" is required in the current directory.

cfg8c Error : bad start-up-file(XXXX)

There is unnecessary start-up-file in the current directory.

cfg8c Error : no source file

No source file is found in the current directory.

cfg8c Error : zero divide error near line xxx (xxxx.cfg)

A zero divide operation occurred in some arithmetic expression.

cfg8c Error : task[X].stack_size must set XX or more near line xxx (xxxx.cfg)

You must set more than XX bytes.in task[x].stack_size.

cfg8c Error : “R0” and “R1” must exist in task[x].context near line xxx (xxxx.cfg)

You must select R0 and R1 register in task[x].context.

cfg8c Error : can’t define address match interrupt definition for Task Pause Function near line xxx (xxxx.cfg)

Another interrupt is defined in interrupt vector definition needed by Task Pause Function.

cfg8c Error : Set system timer [system.timeout = YES] near line xxx (xxxx.cfg)

Set clock.timer symbol except “NOTIMER”.

cfg8c Error : Initial Start Task not defined

No initial startup task is defined in the configuration file.

Warning messages

The following message are a warning. A warning can be ignored providing that its content is understood.

cfg8c Warning : system is not defined (xxxx.cfg)

cfg8c Warning : system.XXXX is not defined (xxxx.cfg)

System definition or system definition item XXXX is omitted in the configuration file.

cfg8c Warning : task[x].XXXX is not defined near line xxx (xxxx.cfg)

The task definition item XXXX in ID number is omitted.

cfg8c Warning : Already definition XXXX near line xxx (xxxx.cfg)

XXXX has already been defined.The defined content is ignored, check to delete the extra definition.

cfg8c Warning : interrupt_vector[x]'s default is not defined (default.cfg)

The interrupt vector definition of vector number x in the default configuration file is missing.

cfg8c Warning : interrupt_vector[x]'s default is not defined near line xxx (xxxx.cfg)

The interrupt vector of vector number x in the configuration file is not defined in the default configuration file.

cfg8c Warning : system.stack_size is an uneven number near line xxx
cfg8c Warning : task[x].stack_size is an uneven number near line xxx

Please set even size in system.stack_size or task[x].stack_size.

9. Sample Program Description

9.1 Overview of Sample Program

As an example application of MR8C/4, the following shows a program that outputs a string to the standard output device from one task and another alternately.

Table 9.1 Functions in the Sample Program

Function Name	Type	ID No.	Priority	Description
main()	Task	1	1	Starts task1 and task2.
task1()	Task	2	2	Outputs "task1 running."
task2()	Task	3	3	Outputs "task2 running."
cyh1()	Handler	1		Wakes up task1().

The content of processing is described below.

- The main task starts task1, task2, and cyh1, and then terminates itself.
- task1 operates in order of the following.
  1. Gets a semaphore.
  2. Goes to a wakeup wait state.
  3. Outputs "task1 running."
  4. Frees the semaphore.
- task2 operates in order of the following.
  1. Gets a semaphore.
  2. Outputs "task2 running."
  3. Frees the semaphore.

cyh1 starts every 100 ms to wake up task1.

9.2 Program Source Listing

```
 1 /************************************************************************
 2 *                       MR8C/4  smaple program
 3 *
 4 * COPYRIGHT(C) 2003(2005) RENESAS TECHNOLOGY CORPORATION
 5 * AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
 6 *
 7 *
 8 *     $Id: demo.c,v 1.2 2005/06/15 05:29:02 inui Exp $
 9 ************************************************************************/
10
11 #include <itron.h>
12 #include <kernel.h>
13 #include "kernel_id.h"
14 #include <stdio.h>
15
16
17 void main( VP_INT stacd )
18 {
19     sta_tsk(ID_task1,0);
20     sta_tsk(ID_task2,0);
21     sta_cyc(ID_cyh1);
22 }
23 void task1( VP_INT stacd )
24 {
25     while(1){
26             wai_sem(ID_sem1);
27             slp_tsk();
28             printf("task1 running\n");
29             sig_sem(ID_sem1);
30     }
31 }
32
33 void task2( VP_INT stacd )
34 {
35     while(1){
36             wai_sem(ID_sem1);
37             printf("task2 running\n");
38             sig_sem(ID_sem1);
39     }
40 }
41
42 void cyh1( VP_INT exinf )
43 {
44     iwup_tsk(ID_task1);
45 }
46
```

9.3 Configuration File

```
 1 //************************************************************************
 2 //
 3 //   COPYRIGHT(C) 2009 RENESAS TECHNOLOGY CORPORATION
 4 //   AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
 5 //
 6 //      MR8C/4 System Configuration File.
 7 //        "$Id: smp.cfg,v 1.5 2005/06/15 05:41:54 inui Exp $"
 8 //
 9 //************************************************************************
10
11 // System Definition
12 system{
13       stack_size     = 1024;
14     priority        = 10;
15       system_IPL     = 4;
16     tic_nume        = 1;
17     tic_deno        = 1;
18 };
19 //System Clock Definition
20 clock{
21     mpu_clock              = 20MHz;
22       timer                = A0;
23       IPL                  = 4;
24 };
25 //Task Definition
26 //
27 task[]{
28     entry_address   = main();
29     name            = ID_main;
30       stack_size     = 100;
31     priority        = 1;
32     initial_start   = ON;
33 };
34 task[]{
35     entry_address   = task1();
36     name            = ID_task1;
37       stack_size     = 500;
38     priority        = 2;
39 };
40 task[]{
41     entry_address   = task2();
42     name            = ID_task2;
43     stack_size      = 500;
44     priority        = 3;
45 };
46
47 semaphore[]{
48     name            = ID_sem1;
49     max_count       = 1;
50     initial_count   = 1;
51 };
52
53 cyclic_hand [1] {
54     name                    = ID_cyh1;
55     interval_counter        = 100;
56     start                   = OFF;
57     phsatr                  = OFF;
58     phs_counter             = 0;
59     entry_address           = cyh1();
60     exinf                   = 1;
61 };
```

10. Stack Size Calculation Method

10.1 Stack Size Calculation Method

The MR8C/4 provides two kinds of stacks: the system stack and the user stack. The stack size calculation method differ between the stacks.

- User stack

  This stack is provided for each task. Therefore, writing an application by using the MR8C/4 requires to allocate the stack area for each stack.

- System stack

  This stack is used inside the MR8C/4 or during the execution of the handler.

  When a task issues a service call, the MR8C/4 switches the user stack to the system stack. (See Figure 10.1 System Stack and User Stack

)

  The system stack uses interrupt stack(ISP).

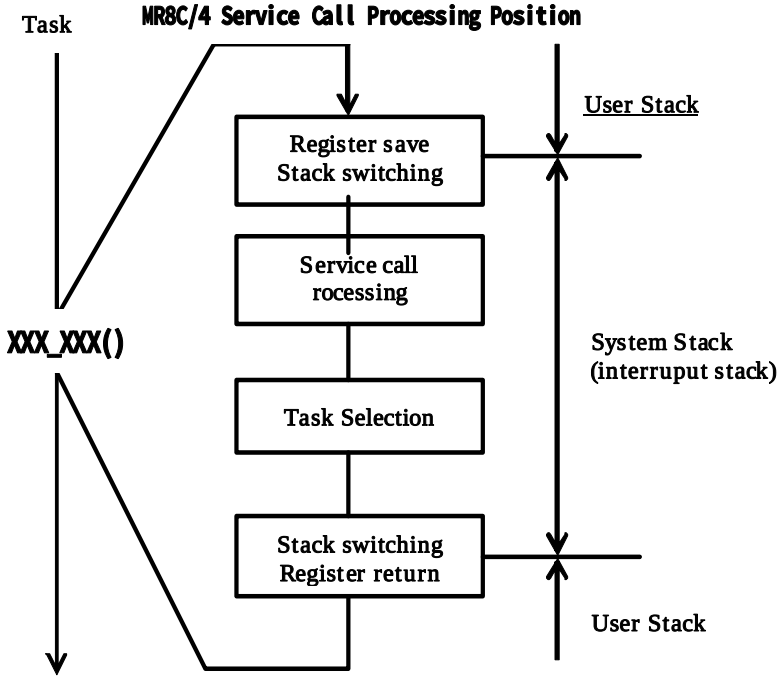


Figure 10.1 System Stack and User Stack

The sections of the system stack and user stack each are located in the manner shown below. However, the diagram shown below applies to the case where the stack areas for all tasks are located in the stack section during configuration.

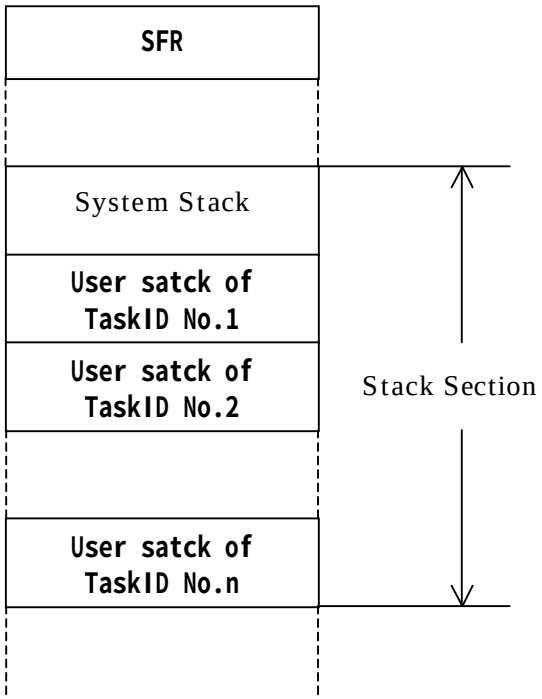


Figure 10.2 Layout of Stacks

10.1.1 User Stack Calculation Method

User stacks must be calculated for each task. The following shows an example for calculating user stacks in cases when an application is written in the C language and when an application is written in the assembly language.

- When an application is written in the C language

  Using the stack caliculation utility, calculate the stack size of each task. The necessary stack size of a task is the sum of the stack size output by STK Viewer plus a context storage area of 20 bytes[37] The following shows how to calculate a stack size using

- When an application is written in the assembly language

User stack size =
Sections used in user program + size of registers used (size of registers which are written as task.context in .cfg file + 6 bytes(PC+FLG register size) + Sections used in MR8C/4

♦ **Sections used in user program**

The necessary stack size of a task is the sum of the stack size used by the task in subroutine call plus the size used to save registers to a stack in that task.

♦ **Sections used in MR8C/4**

The sections used in MR8C/4 refer to a stack size that is used for the service calls issued.

When issuing multiple service calls, include the maximum value of the stack sizes used by those service calls as the sections used by MR8C/4 as you calculate the necessary stack size.

Figure 3.1 shows an example for calculating a user stack. In the example below, the registers used by the task are R0, R1, R2 and A0.

[37] If written in the C language, this size is fixed.

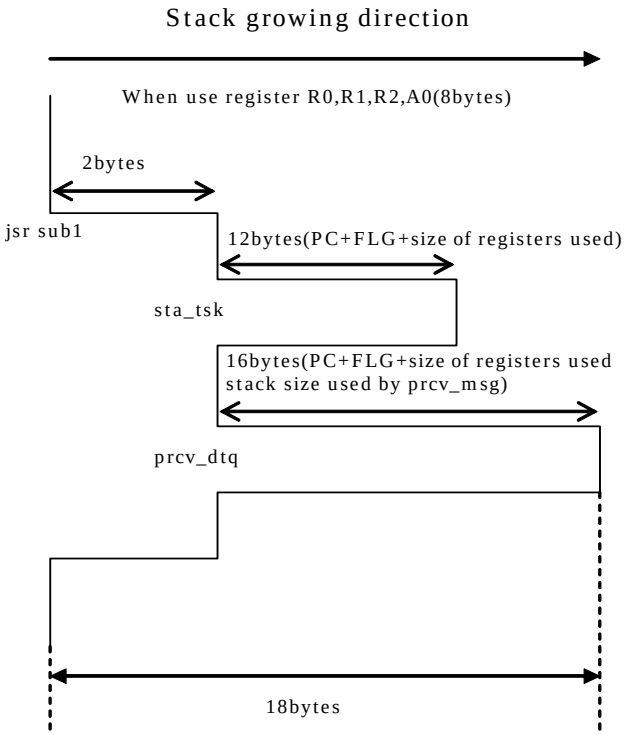


Figure 10.3 Example of Use Stack Size Calculation

10.1.2 System Stack Calculation Method

The system stack is most often consumed when an interrupt occurs during service call processing followed by the occurrence of multiple interrupts.[38] The necessary size (the maximum size) of the system stack can be obtained from the following relation:

$$\text{Necessary size of the system stack} = \alpha + \Sigma\beta i(+ \gamma)$$

- α

  The maximum system stack size among the service calls to be used.[39].

  When sta_tsk, ext_tsk, slp_tsk and dly_tsk are used for example, according to the Table 10.1 Stack Sizes Used by Service Calls Issued from Tasks (in bytes),each of system stack size is the following.

Service Call name	System Stack Size
sta_tsk	2bytes
ext_tsk	0bytes
slp_tsk	2bytes
dly_tsk	4bytes

  Therefore,the maximum system stack size among the service calls to be used is the 8 bytes of dly_tsk.

- βi

  The stack size to be used by the interrupt handler.[40] The details will be described later.

- γ

  Stack size used by the system clock interrupt handler. This is detailed later.

[38] After switchover from user stack to system stack

[39] Refer from Table 10.1 Stack Sizes Used by Service Calls Issued from Tasks (in bytes) to Table 10.3 Stack Sizes Used by Service Calls Issued from Tasks and Handlers (in bytes) for the system stack size used for each individual service call.

[40] OS-dependent interrupt handler (not including the system clock interrupt handler here) and OS-independent interrupt handler.

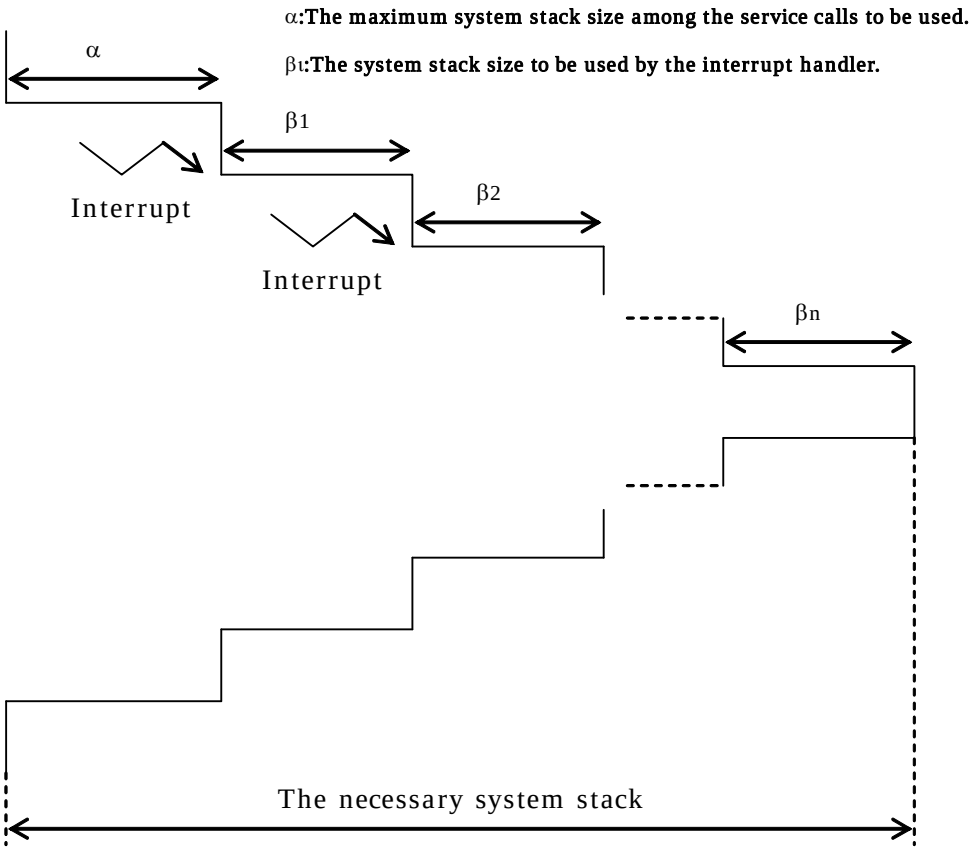


Figure 10.4 System Stack Calculation Method

[(Stack size βi used by interrupt handlers)]

The stack size used by an interrupt handler that is invoked during a service call can be calculated by the equation below.

The stack size βi used by an interrupt handler is shown below.

- C language

  Using the stack caliculation utility, calculate the stack size of each interrupt handler.

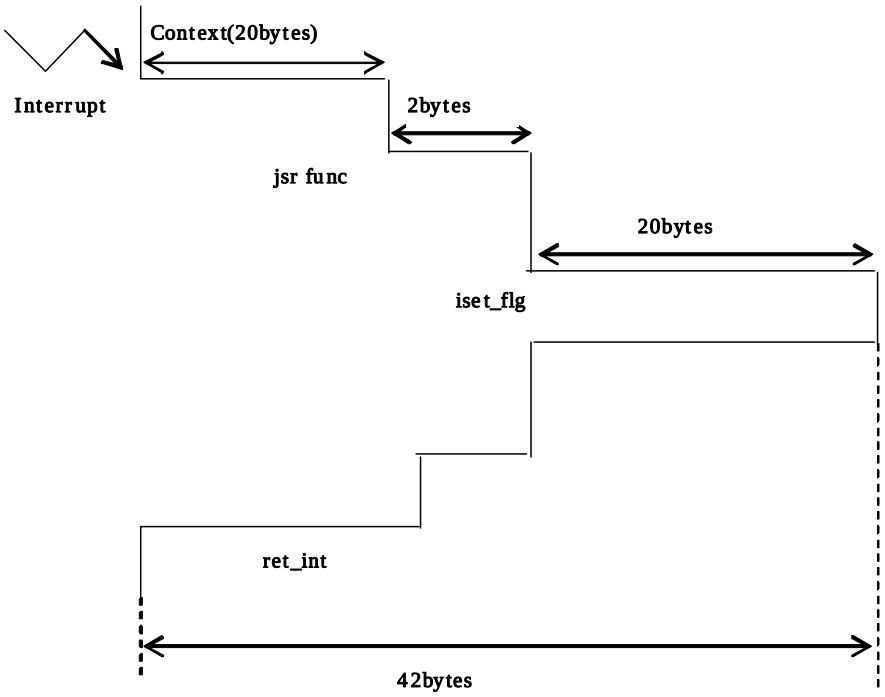
  Refer to the manual of STK Viewer for detailed use of STK Viewer.

- Assembly language

  The stack size to be used by OS-dependent interrupt handler
  = register to be used + user size + stack size to be used by service call

  The stack size to be used by OS-independent interrupt handler
  = register to be used + user size

User size is the stack size of the area written by user.



Context: 20 bytes when written in C language.
When written in assembly language,
Context = size of registers written as task.context in .cfg file
+ 4(PC+FLG)bytes

Figure 10.5 Stack size to be used by Kernel Interrupt Handler

[(System stack size γ used by system clock interrupt handler)]

When you do not use a system timer, there is no need to add a system stack used by the system clock interrupt handler.

The system stack size γ used by the system clock interrupt handler is whichever larger of the two cases below:

- 24 + maximum size used by cyclic handler
- 24 + maximum size used by alarm handler
-
- C language
  Using the stack caliculation utility, calculate the stack size of each Alarm or Cyclic handler.

- Assembly language
  The stack size to be used by Alarm or Cyclic handler
  = register to be used + **user size** + **stack size to be used by service call**

If neither cyclic handler nor alarm handler is used, then

γ = 14bytes

When using the interrupt handler and system clock interrupt handler in combination, add the stack sizes used by both.

10.2 Necessary Stack Size

Table 10.1 Stack Sizes Used by Service Calls Issued from Tasks (in bytes) lists the stack sizes (system stack) used by service calls that can be issued from tasks.

Table 10.1 Stack Sizes Used by Service Calls Issued from Tasks (in bytes)

Service call	Stack size		Service call	Stack size	
	User stack	System stack		User stack	System stack
sta_tsk	0	2	sta_cyc	10	0
ext_tsk	0	0	stp_cyc	10	0
ter_tsk	0	4	sta_alm	10	0
chg_pri	0	4	stp_alm	10	0
slp_tsk	0	2	get_tid	10(5)	0
wup_tsk	0	2	loc_cpu	4	0
can_wup	10	0	unl_cpu	0	0
rel_wai	0	4	ref_ver	12	0
sus_tsk	0	2	dis_dsp	4	0
rsm_tsk	0	2	ena_dsp	0	0
dly_tsk	0	4	snd_dtq	0	4
sig_sem	0	2	psnd_dtq	0	2
wai_sem	0	2	rcv_dtq	(5)	2
pol_sem	10	0	prcv_dtq	(5)	2
set_flg	0	6			
clr_flg	10	0			
wai_flg	(5)	2			
pol_flg	10(5)	0			

(): Stack sizes used by service call in C programs.

Table 10.2 Stack Sizes Used by Service Calls Issued from Handlers (in bytes) lists the stack sizes (system stack) used by service calls that can be issued from handlers.

Table 10.2 Stack Sizes Used by Service Calls Issued from Handlers (in bytes)

Service call	Stack size	Service call	Stack size
iwup_tsk	14	iset_flg	22
irel_wai	14	ipsnd_dtq	6
isig_sem	4	ret_int	10
ista_tsk	14		

(): Stack sizes used by service call in C programs.

Table 10.3 Stack Sizes Used by Service Calls Issued from Tasks and Handlers (in bytes) lists the stack sizes (system stack) used by service calls that can be issued from both tasks and handlers. If the service call issued from task, system uses user stack. If the service call issued from handler, system uses system stack.

Table 10.3 Stack Sizes Used by Service Calls Issued from Tasks and Handlers (in bytes)

Service call	Stack size	Service call	Stack size
sns_ctx	10	sns_loc	10
sns_dsp	10		

11. Note

11.1 The Use of INT Instruction

MR8C/4 has INT instruction interrupt numbers reserved for issuing service calls as listed in Table 11.1 Interrupt Number Assignment. For this reason, when using software interrupts in a user application, do not use interrupt numbers 63 through 48 and be sure to use some other numbers.

Table 11.1 Interrupt Number Assignment

Interrupt No.	Service calls Used
32	Service calls that can be issued from only task context
33	Service calls that can be issued from only non-task context. Service calls that can be issued from both task context and non-task context.
34	ret_int service call
35	dis_dsp service call
36	loc_cpu, iloc_cpu service call
37	ext_tsk service call
38	Reserved for future extension
39	Reserved for future extension
40	Reserved for future extension

11.2 The Use of registers of bank

The registers of bank is 0, when a task starts on MR8C/4.

MR8C/4 does not change the registers of bank in processing kernel.

You must pay attention to the followings.

- Don’t change the regisers of bank in processing a task.
- If an interrupt handler with regisers of bank 1 have multiple interrupts of an interrupt handler with regisers of bank 1 , the program can not execute normally.

11.3 Regarding Delay Dispatching

MR8C/4 has four service calls related to delay dispatching.

- dis_dsp
- ena_dsp
- loc_cpu
- unl_cpu

The following describes task handling when dispatch is temporarily delayed by using these service calls.

1. **When the execution task in delay dispatching should be preempted**

   While dispatch is disabled, even under conditions where the task under execution should be preempted, no time is dispatched to new tasks that are in an executable state. Dispatching to the tasks to be executed is delayed until the dispatch disabled state is cleared. When dispatch is being delayed.

   - Task under execution is in a RUNNING state and is linked to the ready queue
   - Task to be executed after the dispatch disabled state is cleared is in a READY state and is linked to the highest priority ready queue (among the queued tasks).

2. **Precautions**

   - No service call (e.g., slp_tsk, wai_sem) can be issued that may place the own task in a wait state while in a state where dispatch is disabled by dis_dsp, loc_cpu or iloc_cpu.
   - ena_dsp and dis_dsp cannot be issued while in a state where interrupts and dispatch are disabled by loc_cpu, iloc_cpu.
   - Disabled dispatch is re-enabled by issuing ena_dsp once after issuing dis_dsp several times.

   The above status transition can be summarized in Table 3.3.

11.4 Regarding Initially Activated Task

MR8C/4 allows you to specify a task that starts from a READY state at system startup. This specification is made by setting the configuration file.

Refer to 8.1.2 for details on how to set.

12. Appendix

12.1 Assembly Language Interface

When issuing a service call in the assembly language, you need to use macros prepared for invoking service calls.

Processing in a service call invocation macro involves setting each parameter to registers and starting execution of a service call routine by a software interrupt. If you issue service calls directly without using a service call invocation macro, your program may not be guaranteed of compatibility with future versions of MR8C/4.

The table below lists the assembly language interface parameters. The values set forth in μITRON specifications are not used for the function code.

Task Management Function

ServiceCall	INTNo.	Parameter					ReturnParameter	
		FuncCode R0	R1	R3	A0	A1 FuncCode	R0	A0
ista_tsk	33	8	stacd	-	tskid	-	ercd	-
sta_tsk	32	6	stacd	-	tskid	-	ercd	-
ter_tsk	32	10	-	-	tskid	-	ercd	-
chg_pri	32	12	-	tskpri	tskid	-	ercd	-
ext_tsk	37	-	-	-	-	-	-	-

Task Dependent Synchronization Function

ServiceCall	INTNo.	Parameter					ReturnParameter
		FuncCode R0	R1	R3	A0	A1 FuncCode	R0
slp_tsk	32	22	-	-	-	-	ercd
wup_tsk	32	26	-	-	tskid	-	ercd
iwup_tsk	33	28	-	-	tskid	-	ercd
can_wup	33	30	-	-	tskid	-	wupcnt
sus_tsk	32	36	-	-	tskid	-	ercd
rsm_tsk	32	40	-	-	tskid	-	ercd
dly_tsk	32	44	tmout	tmout	-	-	ercd
rel_wai	32	32	-	-	tskid	-	ercd
irel_wai	33	34	-	-	tskid	-	ercd

Synchronization & Communication Function

ServiceCall	INTNo.	Parameter						ReturnParameter			
		FuncCode R0	R1	R2	R3	A0	A1 FuncCode	R0	R1	R2	R3
wai_sem	32	50	-	-	-	semid	-	ercd	-	-	-
pol_sem	33	52	-	-	-	semid	-	ercd	-	-	-
sig_sem	32	46	-	-	-	semid	-	ercd	-	-	-
isig_sem	33	48	-	-	-	semid	-	ercd	-	-	-
wai_flg	32	64	wfmode	-	waiptn	flgid	-	ercd	-	flgptn	-
pol_flg	33	66	wfmode	-	waiptn	flgid	-	ercd	-	flgptn	-
set_flg	32	58	-	-	setptn	flgid	-	ercd	-	-	-
iset_flg	33	60	-	-	setptn	flgid	-	ercd	-	-	-
clr_flg	33	62	-	-	clrptn	flgid	-	ercd	-	-	-
snd_dtq	32	72	data	-	-	dtqid	-	ercd	-	-	-
psnd_dtq	32	74	data	-	-	dtqid	-	ercd	-	-	-
ipsnd_dtq	33	76	data	-	-	dtqid	-	ercd	-	-	-
rcv_dtq	32	84	-	-	-	dtqid	-	ercd	data	-	-
prcv_dtq	32	86	-	-	-	dtqid	-	ercd	data	-	-

System Status Management Function

ServiceCall	INTNo.	Parameter		ReturnParameter	
		FuncCode R0	R3	R0	A0
loc_cpu	36	-	-	ercd	-
dis_dsp	35	-	-	ercd	-
ena_dsp	32	150	-	ercd	-
unl_cpu	32	146	-	ercd	-
sns_ctx	33	152	-	ercd	-
sns_loc	33	154	-	ercd	-
sns_dsp	33	156	-	ercd	-
get_tid	33	144	-	ercd	tskid

Time Management Function

ServiceCall	INTNo.	Parameter					ReturnParameter
		FuncCode R0	R1	R3	A0	A1 FuncCode	R0
sta_cyc	33	128	-	-	cycid	-	ercd
stp_cyc	33	130	-	-	cycid	-	ercd
dly_tsk	32	44	tmout	tmout	-	-	ercd
sta_alm	33	134	almtim	almtim	almid	-	ercd
stp_alm	33	136	-	-	almid	-	ercd

System Configuration Management Function

ServiceCall	INTNo.	Parameter	ReturnParameter

Real-time OS for R8C Family
MR8C/4 User's Manual

Publication Date:	August. 1, 2009	Rev.1.00
Published by:	Sales Strategic Planning Div. Renesas Technology Corp.	
Edited by:	Application Engineering Department 1 Renesas Solutions Corp.	

MR8C/4 V.4.00
User's Manual

Renesas Electronics Corporation
1753, Shimonumabe, Nakahara-ku, Kawasaki-shi, Kanagawa 211-8668 Japan

REJ10J2040-0100